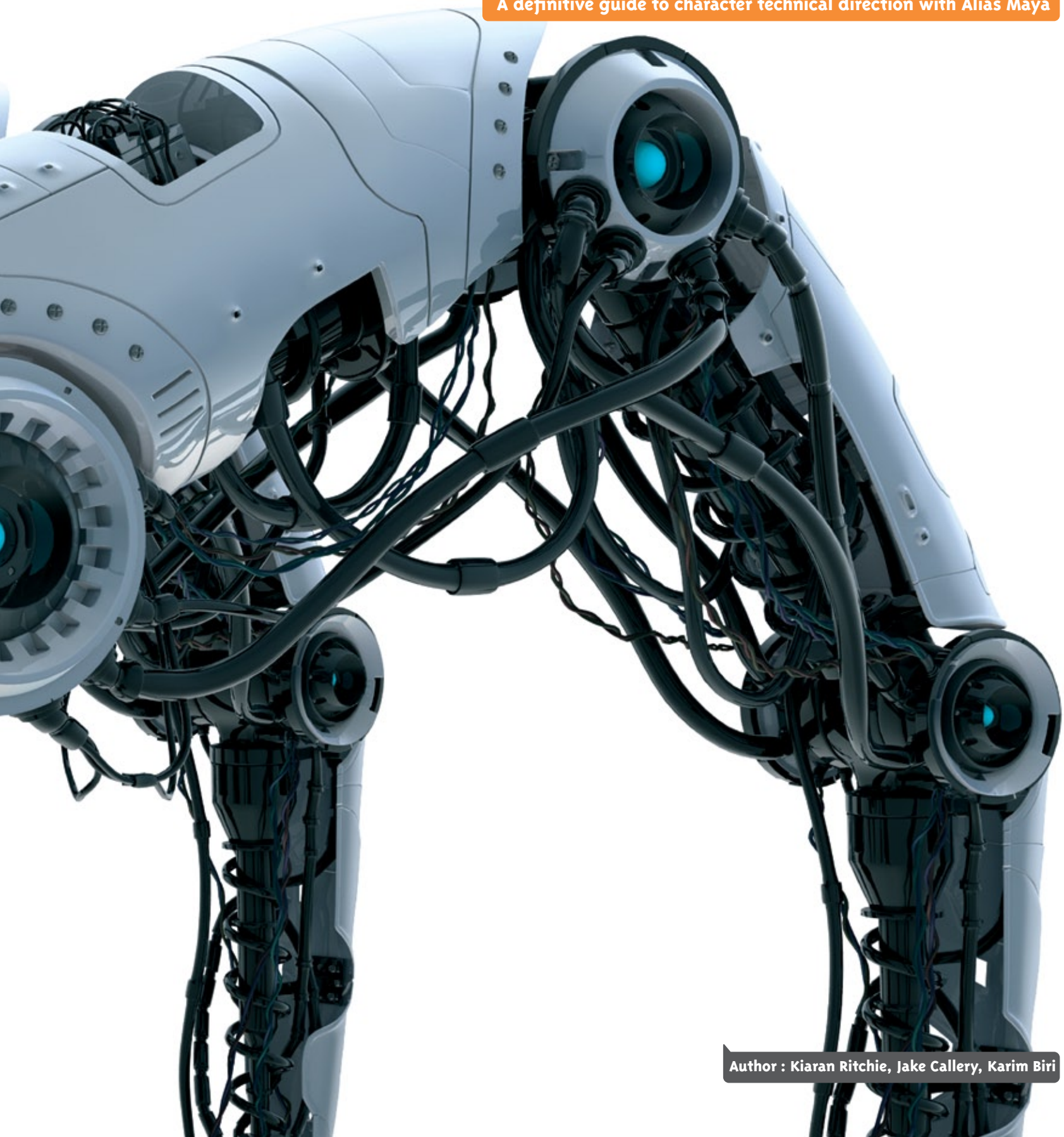
The Art of Rigging

Volume I

A definitive guide to character technical direction with Alias Maya



Author : Kiaran Ritchie, Jake Callery, Karim Biri

Chapter 1
Rigging a Biped

Introduction Bipedel Rigging:

Creature rigging involves many different skills. It is half art and half science. When novice Maya users begin experimenting with joints, constraints, smooth skinning and parenting, they often forget that they are actually programming, albeit with a very high-level language. A finished creature rig, at it's most basic, is simply a large collection of meticulously organized and connected nodes. These nodes form a framework with which the animator can affect a mesh. That is all that a rig does. The sooner you understand this, the sooner you can begin to create creature rigs that are clean, efficient and robust.

Being able to create accurate joint layouts and design good deformations is only part of what a setup artist is responsible for. In a production environment, it is absolutely essential that the finished rig is clean and stable. This chapter is dedicated to showing you how to create creature rigs that will flow through a production pipeline with ease. Those who are new to rigging in Maya will be introduced to the basic kinematic concepts like a reverse foot, no-flip knee and automatic forearm. New users will also be shown how to develop the simplest of MEL scripts, a macro. If you have no (or very little) experience with MEL, this is where to start.

A properly constructed creature should contain a perfectly clean file. Cleanliness and ease of use are of utmost importance. Throughout this book you will hear various ideas that will help you create setups that adhere to these strict conventions. While the creation of a 'perfect' rig file involves many different problems for every different creature, there are some basic rules that are universal. By the end of this chapter, you will be exposed to not only the specifics of what constitutes a 'perfect' rig, but also *why* this makes a rig perfect.

Specifically, this chapter will cover:

- An introduction to Maya's architecture.
- How to use joints.
- IK vs FK
- Iconic Representation
- Cleaning your creature.
- No-Flip IK Knee
- Group-based Reverse Foot
- Auto-Forearm Twist
- Introduction to MEL Scripting

Introduction

So what exactly is 'The Art of Rigging'? This book is a collection of practical theories that have been honed from experience to provide the reader with a full understanding of what it takes to bring believable creatures to life. Rigging is the term used to describe the process of imbuing a lifeless 3d model with the articulation necessary to fool an audience into believing it exists. Creature rigging is a unique field. The best creature riggers are masters of many trades. It is not uncommon that a creature technical director will be responsible for modeling, animating and programming... all in the same day.

The creature technical director can be thought of as the backbone for an entire production. They must have an understanding of exactly how to take a static 3d mesh and turn it into a moving, living creature. At the core of a technical director's job is the task of designing a system of control that allows an animator to do his/her job.

Aside from building great creature rigs, technical directors are often responsible for directing the way the creature flows through an animation pipeline. Because every production is different and the needs of today's productions are as vast as they are varied, technical directors must be particularly adept at creating workflow tools.

This book is your guide through this chaotic, ever-changing field. Rather than writing a 600 page 'Rigging Bible', we have deliberately chosen to make this a practical guide. That is why you will see very little pure theory in this book. It was designed, from the ground-up, with practicality in mind. All of the examples are useful in a production environment. The MEL coverage is designed to show you how to use MEL to create amazing tools. These tools are extremely useful on their own, but their main value comes from the detailed description of their creation. By showing the reader the exact steps to get started writing useful MEL scripts, we hope to leave you with an understanding of not only how useful these tools are, but how you can create your own.

This is a unique 'computer book'. You will find it chalk full of useful tips and techniques that can only come from experience. We understand that most artists are visual learners and they love to see exactly how things work. That is why this book is printed in full color and sprinkled with pictures and screen captures to bring the text to life. The experience of reading this book should be more like that of a very interesting magazine, as opposed to a stale, dusty textbook.

For added value, we have included a DVD that is jam-packed with video lectures. This DVD is not an afterthought. It includes complete coverage of every exercise in the entire book, as well as several lectures on topics across every chapter. These lectures are recorded in high resolution (1024x700) and specially encoded (using Apple's mpeg4 codec) to provide an outstanding learning experience. You can watch as the author guides you through each chapter as though you were looking over his shoulder.

This is an amazing book, inspired by an equally amazing profession. The Art of Rigging, along with some hard work and passion, will enable you to realize your artistic visions. While this book is admittedly Maya-centric, good creatures are built by good artists. The fundamental concepts presented here are universal and should serve you well regardless of what software you use, now or in the future. In the wise words of Michelangelo, "A man paints with his brains and not with his hands".

■ Who Should Read This Book

This book is geared towards those who wish to direct the technical aspects of a digital animation production. These people have many names. In many studios, they are called riggers, but they can also be known as setup artists, technical directors and puppeteers.

While setup artists will get the most value from this book, character animators will find it an excellent way to dig deeper into their craft to discover the many ways that Maya and MEL can be used to improve their work.

Producers will find this book useful for helping them create a smooth, cost-efficient animation pipeline. Technical direction can make or break a production. A poorly directed production will suffer from buggy rigs and inefficiencies that rob time and money. In the end, technical direction can drastically affect the quality of the final, rendered frames.

Lastly, students will find this book a great way to fast-forward their education by learning industry standard techniques. A certain amount of basic Maya knowledge is assumed, but novice Maya users should have no problem understanding the concepts presented in this book. This stuff can be difficult, but the simple writing style and DVD videos can help everyone get a handle on things.

■ Who Should Not Read This Book?

Those who are looking for an introduction to Maya, should keep looking. This book is geared towards users with at least some experience in Maya. If you are coming from a different animation package or have no experience with digital animation at all, I would highly recommend that you take the time to become acquainted with Maya before reading this book.

While many of the topics are presented as clearly as possible, some Maya knowledge is assumed. This is done for the sake of clarity.

■ Chapter Overview

This book is organized into six chapters. Each chapter will tackle various rigging related concepts, while providing step-by-step exercises for some of the more specific setups. In addition, each chapter is finished off with the discussion of a MEL script. These MEL scripts are included on the DVD and will be of great help to you.

■ Chapter 1: Rigging a Biped

This chapter provides an insightful introduction to the rest of the book. After learning how to utilize Maya's unique node-based architecture, this chapter guides the user through several fundamental setup techniques including an IK/FK switch, reverse foot, auto-twist forearm and no-flip IK knee. In addition, you will be introduced to Maya's embedded language. At the end of chapter one, the reader is guided through the complete creation of an auto-rigging MEL script. This script will automatically create an IK/FK switch and a reverse-foot rig.

■ Chapter 2: Squash and Stretch

Chapter two introduces an important concept in digital animation. Readers will be guided through the creation of stretchy joint chains using a MEL expression, utility nodes and finally a custom MEL script. The principles of squash and stretch are applied to a spine and explained in an exercise to show exactly how to rig a stretchy spine. Finally, the reader is guided through a full description of a script that will automatically create a stretchy joint chain.

■ Chapter 3: Overlapping Action

The third chapter introduces the important concept of automatic secondary motion. This chapter describes how to create dynamic joint chains that will flop around to give your creations an element of realism. Maya's dynamics are applied to joint chains using Maya Hair, soft bodies, particle goal weights, the jiggle deformer and a MEL expression. A real production scenario is explored where the reader will be guided through the task of adding antenna and tail animations to a creature's walk cycle. Finally, the user is guided through the complete creation of a MEL script that will automate the process of creating a solid, production worthy dynamic joint chain.

■ Chapter 4: The Face

Chapter four is a massive collection of techniques and tips for creating believable facial rigs. Users are guided through the common face modeling pitfalls. Using the Zuckafa character, the reader will see a full description of Dr. Paul Ekman's action units. These are then used to derive a shape set that will describe every possible facial pose. To create this shape set, users are shown expert modeling techniques. To help the animators, the readers are shown techniques that can be used to control these complex facial rigs. Unique and interesting interface designs are fully discussed. Finally, the user is guided through the creation of a MEL-based pose library system that enables animators to save and share facial poses via pose files. These poses can then be set with the click of a button.

■ Chapter 5: The Pipeline

Many books make the mistake of ignoring exactly how a creature can easily flow through a production pipeline. The task of getting from animation to final render is not an easy one. This chapter provides you with a production worthy tool (a MEL script) for transferring animation across low/medium/high resolution creature rigs. To help users understand how pipeline tools like this are created, the reader is guided through the creation of the entire MEL script. This chapter includes a very deep explanation of MEL scripting techniques. The concepts used for this utility will be of great help to those who wish to write their own pipeline tools.

■ Chapter 6: Muscles And Fat

To round off the discussion of organic deformations, this chapter tackles the problems associated with designing believable, usable deformations. Those who are familiar with the traditional method of using influence objects will be ecstatic to discover the next revolutionary deformation tool, pose space deformation or PSD. After learning how to sculpt inter-muscle collisions with per-vertex control, the reader will be shown how to apply dynamic fatty jiggle to your creatures. All of this is done using only joints and blendshapes to provide a fast, stable rig capable of export to a game engine. This chapter finishes with a full description of how to create a MEL script that will simulate PSD using blendshapes.

■ Before You Start...

This book assumes some basic experience with Maya. If you do not know what a set driven key is or how to create an IK handle, you may wish to start somewhere else. There are plenty of books that are specifically designed to give the reader a solid understanding of how Maya works. If you are completely new to Maya, I highly recommend that you read one of these first. This will deepen your understanding of the concepts in this book, and reduce the number of head scratching moments.

That being said, if you are ready to dig deeper into your favorite package, get ready, because it's going to be a fun ride! This book can be read in a linear fashion or used as a reference. The MEL examples at the end of each chapter get progressively more difficult. If you are trying to learn how to create better MEL scripts, we recommend that you read the book in a linear fashion. This will introduce the more advanced concepts after the simple ones become second nature.

Please do not forget to watch the included DVD lectures. These were designed to compliment the text and include many extra tips and techniques that are not found in the text. If you are having trouble with an exercise, watch the corresponding lecture to see how the author has solved a particular problem. This results in a far richer learning experience that you are sure to enjoy.

Lastly, please understand that while every caution has been taken to ensure accuracy, every book is susceptible to errors. If you encounter something that seems out of place, please report it to errata@cgtoolkit.com.

■ Dedications:

To my loving Mom and Dad, Karlene and Sheena for supporting my ambitions in every way. To the #303 boys (past and present), for their encouragement. To my lovely girlfriend Jasmine for her inspiring patience.

Kiaran

To my soon to be wife Kim, for pushing me to get this done, and inspiring me to keep at it. To Joseph Newcomer, for your patience and time spent that helped me become who I am today. And to my parents Elaine and Tony for always being there when I needed it.

Jake

To my lovely wife Karine and Daughter Kira, to my grand mother Genevievre, to my parents Danielle and Johnny and my brother Steeve

Karim

■ Contributors

3d models,concept
Kursad Karatas
Phattro
Emrah ELMASLI
Cluster Studios
Juan Carlos Lepe Vázquez
Edgar Fernando Piña Vásquez

Table of Contents

Introduction

Chapter One: Rigging a Biped 3
Maya's Architecture 4
The Dependency Graph 5
The Directed Acyclic Graph 5
Getting Under The Hood 5
How To Use Joints 5
What Are Joints? 5
Placing Joints 6
Finalizing Joints 8
IK vs. FK 9
Iconic Representation 9
Pick Walking 11
Pretty Colors 11
Cleaning Your Creature 11
Dirty Channels 11
Hypergraph Organization 12
Layers and Multi-Resolution Rigs 12
The No-Flip IK Knee 13
Exercise 1.1: The No-Flip Knee 13
The Classic IK/FK Switch 16
Exercise 1.2: Creating a Classic IK/FK Switch 17
The Group-Based Reverse Foot 18
Exercise 1.3: A Group-Based Reverse Foot 18
The Auto-Forearm Twist 21
Exercise 1.4: An Auto-Twist Forearm Setup 21
Introduction to MEL Scripting 24
What is MEL? 24
Why Learn MEL? 25
How Does MEL work? 25
Procedures and Commands 25
Data Types 26
Arrays 26
The cgTkSetupLeg.mel Script 27
The Code 28
Scripting the IK/FK Switch 29
Scripting the Reverse Foot 32
Final Thoughts 35

Chapter Two: Squash and Stretch 36
Introduction to Squash and Stretch in Computer Animation 37
Exercise 2.1: Creating a Stretchy Joint Chain (MEL Expression) 38
Exercise 2.2: Creating a Stretchy Joint Chain (Utility Nodes) 40
Exercise 2.3: Using cgTkStretchyIk.mel to Setup Limbs 42
Exercise 3.4: Rigging a Stretchy IK/FK Spine 45
Mathematics and MEL 49
The Distance Formula 50
The Creation of cgTkStretchyIk.mel 50
The 'makeIkStretchy' Procedure 51
Final Thoughts 54

Chapter Three: Automatic Overlapping Action 56
Introduction to Overlapping Action in Computer Animation 57
Maya Unlimited's Hair Dynamics for Overlapping Action 58
Exercise 3.1: Creating a Dynamic Joint Chain (Hair) 58
Automating Setups With MEL 59
Exercise 3.2: Using cgTkDynChain.mel 60
Dynamic Overlapping Action Without Hair 63
Exercise 3.3: Creating a Dynamic Joint Chain (Springs) 63
Exercise 3.4: Creating a Dynamic Joint Chain (Goal Weights) 65
Exercise 3.5: Creating a Dynamic Joint Chain (Jiggle Deformer) 67
Exercise 3.6: Creating a Dynamic Joint Chain (Expression) 68
MEL Scripting: The Creation of cgTkDynChain.mel 70
Preparation 70
Your Development Environment 71
ELF User Interfaces 71
General Program Structure 71
Starting The Script 71
The UI is Done... For Now 73
Connect The Dynamics 78
Finishing Touches 80
Collisions 80
Baking the Joints 82
Error Handling 84
Chapter Summary 84

Chapter Four: The Face 86
Introduction to Facial Setup and Animation 87
Elements of a Successful Facial Rig 88
Style 88
Blendshape-Based Facial Animation 89
Blendshape Theory 89
Building a 'Bottom-Up' Facial Rig 90
Modeling For Animation 91
Projection Modeling 93
Smoothing 94
Modeling Preparation 95
Defining A Facial Shape-Set 96
Understanding the Character 96
The Human Face 96
All 44 Action Units from the Facial Action Coding System 98
Interpreting Action Units 105
Secondary Actions 105
The Final Shape Set 105
Zuckafa's Shape Set 105
A Final Word On Shape Sets 115
Constructing a Shape Set 115
Preparation and Organization 115
Push and Pull Like Clay 115
Mirroring a Blendshape 116
Exercise 4.1: Sculpting the Smile Shapes 118
Smoothing Skinning the Face 120
Exercise 4.2: Smooth Skinning the Jaw 121
Interfacing With the Animator 125
Higher Levels of Control 125
Exercise 4.3: Rigging Soft Eyes 126
MEL Scripting: The Creation of Zuckafa's Pose Library UI 129
Zuckafa's Pose Library UI 130
File Input/Output (I/O) 130
File Formats 130
Getting Started with the Interface 131
The 'Set Key' and 'Reset' Buttons 133
Coding the Pose Library 134
Writing the Pose To Disk 137
Load Pose Library File Procedure 138
How The Parser Works 138
Wrapping it Up 139
Finishing Touches 139
Embedding a Script 139
Final Thoughts 140

Chapter Five: Streamlining the Animation Pipeline 143
A Case Study in Tool Development 143
Problem Identification 144
User Interface 144
Making a Window 147
Building the UI 151
Creating Functionality for the 'Save' Tab 164
jcSaveAnimFile Procedure 167
Creating Functionality for the 'Load' Tab 182
Creating Functionality for the 'Batch' Tab 198
Final Thoughts 207

Chapter Six: Muscles and Fat 210
Introduction to Muscle and Fat Deformations in Computer Animation 211
A Brief History of Deformations 212
3d in a Nutshell 212
Deformations in a Nutshell 212
Pose Space Deformation 213
Exactly How Awesome PoseDeformerv1.15 Is 215
So What's the Catch? 216
Exercise 6.1: Using Shape Builder v1.0 on Dassak's Knee 217
Defining Complex Poses 221
Single Axis Drivers 221
Twin-Axis Drivers 221
Three-Axes Drivers 222
The Multi-Axis Driver Expression 222
Beautiful, Lovely, Sexy Corrective Shapes 225
What is the Matrix 226
Corrective Shapes Across the Body 227
Jiggly, Wiggly, Flabby, Fatty Flesh 229
Exercise 6.2: Muscle and Fat Jiggle 229
MEL Scripting: The Creation of cgTkShapeBuilder.mel 233
The Shape Builder Interface 233
The Other Procedures 236
The loadBlendshapeNode Procedure 236
The loadBaseMesh Procedure 237
The createSculptDuplicate Procedure 237
The refreshBlendshapeList Procedure 238
The loadDriverJoint Procedure 239
The jointSliderCommand Procedure 239
The shapeSliderCommand Procedure 240
The setShapeKey Procedure 240
The editCurve Procedure 241
The generateBlendshape Procedure 241
The BSpiritCorrectiveShape Script 242
The Math Principles 243
The BspiritCorrectiveShapePosition Procedure 249
Index

Why Does a Rig Need to Be Perfect?

More so than most 3d animation disciplines, rigging is founded on principles from math and science. While modelers strive to make their meshes artistically correct, they have relatively few technical problems to worry about. The same can be said of animators and texture artists. For the most part, constructing a rig is more like programming than anything else. Rather than using a computer language, a rigger uses the hypergraph in conjunction with Maya's kinematics and deformation tools to build a system that defines the motion of a mesh. So while there are many artistic considerations to be made, a creature rig absolutely must behave properly. In the same way that a sloppy programmer will create buggy software, a sloppy rigger will create buggy rigs.

Do not try to pass a rig down a production pipeline if you think it has bugs in it. Chances are it will end up creating more work for you than if you had just taken the time to perfect it in the first place. While the most non-linear of animation pipelines will allow for some measure of post-animation rig tweaking, there are some problems that are utterly devastating and will result in the complete loss of animation. As a creature rigger, it is your job to catch these bugs *before* they wreak havoc on your production.

Unfortunately, there is no magic 'rig debugger' or any such thing. Finding and eliminating these problems is entirely up to you. With lots of practice (and some help from this chapter), you will be able to find 99% of the potential bugs before they do any harm. Please know that the remaining 1%, is just bad luck. No matter how diligent you are, problems will always arise. Do not worry, this happens to everybody. Deal with these as they arise and try not to get discouraged. I have found that the best TDs are problem solvers. If a rig blows-up in your face, take a deep breath and work through the possible solutions. Usually the problem is far less serious than you think. As you gain experience, these problems will bother you less and less.

To help you understand how a rig actually works (along with everything else in Maya), let's start with a discussion about exactly how Maya works at it's very core, the dependency graph.

Maya's Architecture:

This section is not about designing buildings with Maya. In software engineering lingo, an application's architecture is described as the way the software's elements interact to affect an outcome. Maya's architecture is one of the best in the industry. It is extremely extensible and open to adaptation. This is why it has become the 3d animation package of choice for the demanding film and television industries.

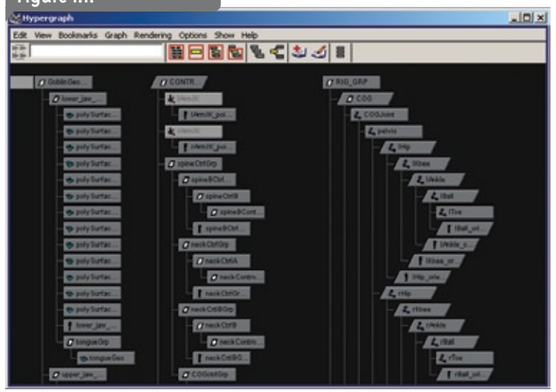
On a less abstract level, Maya's architecture can be described as being node based. A Maya scene file is comprised of a series of nodes that are connected. Everything in Maya is a node, and the effect of these nodes is determined by the way they are connected.

When you create objects, constraints, keyframes or shaders, Maya is arranging the necessary nodes and creating the proper connections. It does this without being told to. In fact, almost every tool and button in the Maya interface is designed to affect a node or it's connections in some way. For most tasks, it is not even necessary to be aware of exactly what Maya is doing in the background. Because Maya is designed to facilitate some common tasks (like modeling, texturing and animating), many people can build a successful career with Maya without even being aware of it's node-based architecture.

While the Alias software engineers have done a wonderful job of providing quick access to the most commonly used processes, they realized that they could not account for every need. Many tasks require that the user have an understanding of how the underlying node-based system works. Because creature rigging encompasses so many different possibilities, technical directors must be particularly adept at making their own node connections.

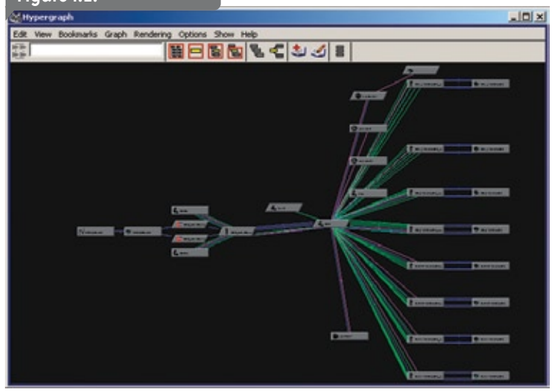
There are two basic types of nodes in Maya. They are DAG and DG nodes. DAG FIGURE 1.1:

Figure 1.1.



A DAG or, Directed Acyclic Graph, node is one that exists within the hypergraph and outliner. These include transform, shape, constraints and many others. Transform nodes contain the transformation information of an object and are edited from the translate, rotate and scale attributes in the channel box. Shape nodes are parented under transform nodes and contain the transformation information of the components of an object (like vertices of a mesh or CV's of a NURBS curve). The main difference between a DAG and a DG node is that DAG nodes must be arranged in linear parent/child hierarchies. You are not allowed to create cycles (hence the acyclic part of the name).
DG FIGURE 1.2:

Figure 1.2.



A DG or 'Dependency Graph' node is one that is not displayed in the outliner. These nodes includes keyframes, shaders, utility nodes, deformers and much more. The main different between a DG node and a DAG node is that DG nodes can be connected in cycles. You may have noticed that Maya sometimes returns a warning about DG nodes that have been connected in a cycle.

Warning: Cycle on 'pCube1_parentConstraint1.target[0].targetRotate' may not evaluate as expected.

While Maya does allow this to happen, it cannot guarantee that the results will be predictable. In general, DG nodes are well hidden from the user and rarely require direct manipulation (with some exceptions).

To understand exactly how Maya's node-based architecture works, let's start by taking a look at it!

■ The Dependency Graph

If all this 'node architecture' techno-babble sounds intimidating, relax. It is really quite friendly once you see it in action. Maya provides a great way for you to see it's underlying magic in something called the dependency graph. A dependency graph can be viewed for any object to give a graphical representation of all up/downstream connections. To view the dependency graph for an object, select it in the hypergraph and hit the 'Input/Output Connections' button FIGURE 1.3. This will display a graph of connected nodes. The nodes are displayed as small boxes with the connections displayed as lines connecting the boxes.

Figure 1.3.

In a dependency graph, you can follow the flow of data by following the connections between the nodes. You may wonder then, does the data flow from left to right? The answer is, not always. Nodes in the dependency graph can be connected in such a way that they form a cycle. This is usually (though not always), a bad thing. Cycles can be thought of as infinite loops. They can produce buggy unpredictable results.

When everything is working as it should, data can flow through a chain of nodes with each node affecting the data in some way to produce a final output. This is another way of expressing cause and effect. When one upstream node's data is changed, the effect can propagate through the downstream nodes to affect the final output.

There are many ways to make a connection between dependency graph nodes. The connection editor, set driven keys, expressions, hypershade and MEL can all edit connections between nodes. In addition to these tools, you can make connections directly in the hypergraph while viewing the dependency graph. To do so, simply right-click on the right side of a node to bring up a list of outputs. With an output selected, you can then click on the left side (input side) of a different node to bring up a list of inputs. Clicking on an input will create the connection that is drawn as a line with an arrow pointing in the direction of the data flow FIGURE 1.4.

Figure 1.4.

■ The Directed Acyclic Graph

This is also known as the scene hierarchy or DAG view. The DAG is displayed in Maya's outliner and hypergraph windows. It provides an organized view of all the objects in your scene. When you select an object in the viewport, you are selecting a DAG object.

When you open the hypergraph, you are treated to a view of the contents of your scene as they relate to each other. Parents are at the top, and children are displayed in branches underneath. The hypergraph provides an excellent way of intuitively creating complex hierarchies like those found in a creature rig. The DAG view, in the hypergraph, is without a doubt, the setup artists most important window.

To edit the parent/child relationships of your scene's objects, simply middle mouse drag objects in the hypergraph. This provides an excellent way of quickly editing joint hierarchies in a creature rig.

■ Getting Under the Hood

To take full advantage of the power that Maya offers, you must have a solid understanding of how it works. Maya's architecture is one of the best in the industry. By understanding exactly how the DAG and DG work together to describe a scene, you are free to edit it however you like.

If you decide to dive deeper to take advantage of Maya's extensibility (by writing scripts and plugins), your understanding of these graphs will deepen. As you gain experience rigging creatures, the DAG will become second nature. For now, at least be aware that these graphs exist as a way of controlling *everything* in your scene.

How To Use Joints:

What are Joints?

Every 3d animation package contains one of two different features for describing a hierarchy of control. These are either joints or bones. While 3ds max uses bones, Maya uses joints. The difference is simple. A joint is simply a point in the scene's space about which things can pivot. A bone is two joints connected so that one is a child of the other. To create a bone in Maya, you simply create two joints, with one being a child of the other FIGURE 1.5.

Figure 1.5.

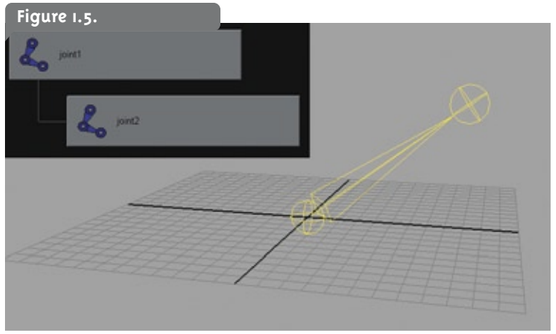
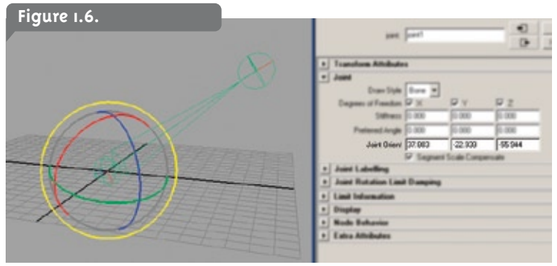
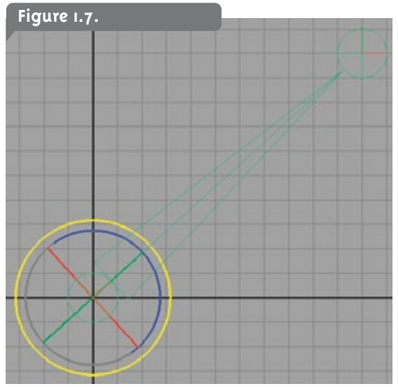


Figure 1.6.



Joints, in Maya, are a unique type of transform node. They have the ability to be rotated to point in a direction that is different from their orientation. Their local rotation axis can be oriented to any arbitrary direction (like world space), while the bone is pointing in a completely different direction FIGURE 1.6. The task of aligning the local rotations axis of your joints is known as *orienting* the joints. When you create a joint chain, Maya will automatically try to orient your joints such that the local rotation axis is aligned with the child joint. By default, this creates a behavior where the X-axis becomes the twisting axis, the Z-axis rotates the joint up and down and Y, side-to-side. This is known as an XYZ orientation (where the first axis, X, becomes the twisting axis) FIGURE 1.7. This is the orientation convention that we will be using in this book. Some setup artists prefer to orient their joints such that Y becomes the twisting axis. This is often done to accommodate motion capture data. Unless you have a specific reason to use a different default orientation, I recommend sticking to the default XYZ.

Figure 1.7.



In the case of creature rigs, it is often handy to orient your joints by hand. For hinge joints (those that only have one axis, like a knee or elbow), it is best to ensure that one axis of the Maya joint is pointing in the proper direction. This allows you to lock the other axes and prevent the animator from breaking the joint by rotating it in an impossible axis (more on locking attributes later). By default, Maya may have oriented the joint such that the local rotation axis does not point in the correct direction that you want the hinge to rotate from. In these cases, you can edit the local rotation axis by hand to point in the proper direction. To do this, click the 'Component Mode' button in the top status line. Then, click on the '?' icon FIGURE 1.8. Now you can click on a joint and rotate it's local rotation axes. To view the local rotation axis for a joint, select it and execute `Display > Component Display > Local Rotation Axes`.

Figure 1.8.

Placing Joints

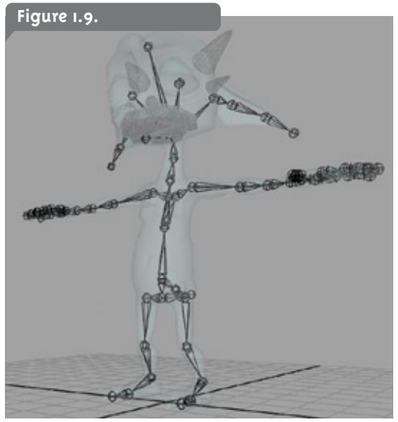
When you are placing joints to build a creature's skeleton, there are a couple of considerations to be made. The most obvious question is:

How many should you use and where exactly should they be placed?

To make your life much easier, it is best to forget anatomical accuracy right away. There are 206 bones in a human skeleton, more than half of which are in your hands and feet. Accurately reproducing every single joint would be an exercise in futility. Not only would this be a major pain in the ass to setup, it would be impossible to animate. So rather than trying to simulate the human skeleton's bone structure, setup artists try to simulate the *range of motion*. By splitting the body into clearly defined hierarchies, we can accurately simulate the way a real skeleton moves, without worrying about every little bone and muscle. Remember that the ultimate purpose of your rig is to affect a mesh in a believable manner. The audience will not care if you took a week to meticulously place every vertebrae joint. They only care that the creature looks and moves in a believable manner.

While every setup artist seems to have their own joint placement conventions, everybody agrees on at least a basic layout for a biped creature. This layout includes articulation for all of the main limbs, the spine, neck, head, jaw, feet and hands FIGURE 1.9. For this chapter, we will be examining the layout for a bipedal goblin creature. While he is certainly not human, he is a biped, and as such, requires the same joint layout.

Figure 1.9.



Let's dissect the joint layout for the Goblin. Our Goblin creature has three joints starting from his pelvis and working up to his neck. For most creatures, 3-5 spine joints provides enough resolution to pose the spine into any position FIGURE 1.10. Some people prefer to use many more (10-20) to simulate individual vertebrae. This is fine, but it requires a spine setup that squishes the joints into about 3-5 different controllers. I usually name these joints COG (Center Of Gravity), spineA, spineB, spineC etc...

Figure 1.10.

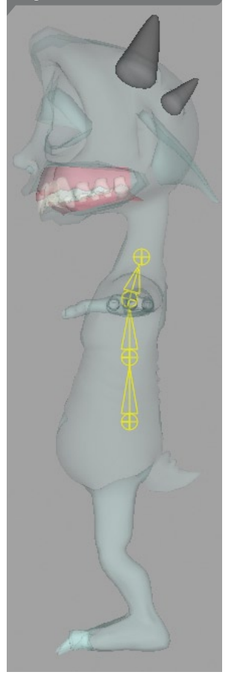
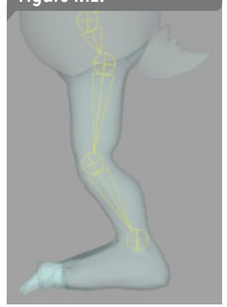


Figure 1.11.

Figure 1.12.



To provide a separate control for the hip region, it can be handy to create another joint directly below the COG FIGURE 1.11. This joint can be rotated to provide the swinging Elvis-hip type motion. Be sure to parent the leg chains under this hip joint.

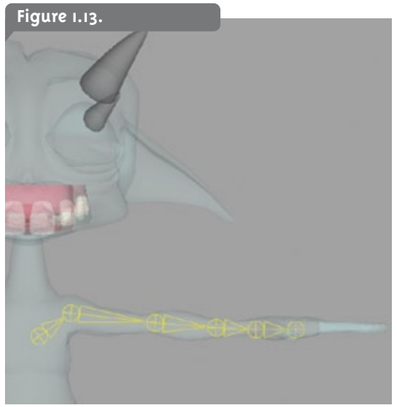
Moving down the body, the Goblin has one joint at the hip, knee and ankle FIGURE 1.12. The hip joint is a ball and socket and so should be free to rotate in all three axes. The knee joint is a hinge with only one free axis. The ankle can be free to rotate in all directions. Moving down the leg, we have two more joints to complete the foot. These are the ball and toe joints. These joints are necessary for a reverse-foot setup (more on this later). You may notice that the Goblin's toes are not individually jointed. For this simple rig, we skipped individual toes. You may wish to add separate toe joints. To do so, simply draw a chain of joints for each toe and parent these chains under the main toe joint. This is all the articulation needed for a regular leg rig.

For the arms of the Goblin, I chose a fairly common setup which includes six main joints. They are the clavicle, shoulder, elbow, forearm, wrist and palm FIGURE 1.13. Arms can be somewhat more tricky than a leg. The clavicle is at the top of the arm chain and is the child of the nearest spine joint. The clavicle bone must be carefully placed to provide a believable radius of motion. The entire trapezius muscle activates the shrugging motion that this joint simulates. To get it to look correct, you may wish to try several placements before moving on to the shoulder.

The shoulder is a ball-in-socket joint with three axes of motion. This joint must be placed as close as possible to the anatomical pivot. Be careful to ensure that it is placed correctly by positioning it in a perspective viewport. The elbow is placed in a similar manner. Try to get it as close as possible to it's anatomical pivot.

The forearm joint is tricky. It is best if this joint lies directly in line with the elbow and the wrist. Holding down the shift key while placing joints can do this, but this will only work if the creature's arm is modeled in a perfectly straight line. To help you place the forearm joint, you

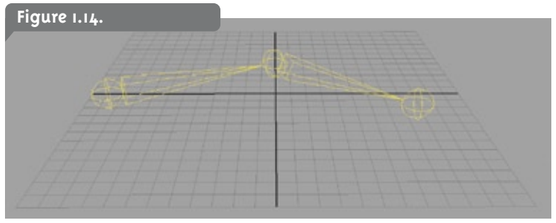
Figure 1.13.



can use this simple trick:

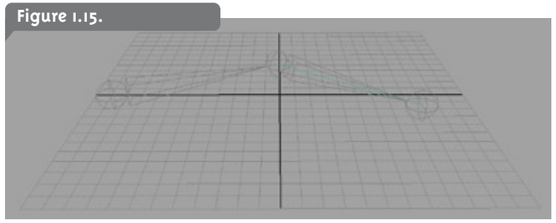
1. Place the shoulder, elbow and the wrist joints first. This will create two bones. The forearm joint must lie directly between the second and third joint, on the forearm bone. FIGURE 1.14.

Figure 1.14.



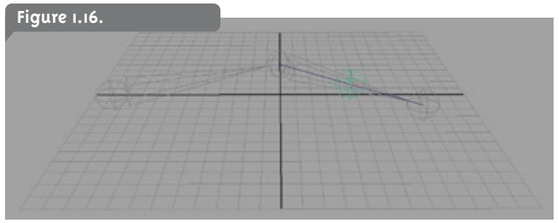
2. To place a joint directly between the elbow and the forearm, will will create a straight curve between the joints and point snap the new forearm joint to this curve. Activate the EP curve tool and create a 1 degree curve with the first CV point snapped (using the 'V' hotkey) to the elbow and the last CV point snapped to the wrist. Press enter to finish the curve. This creates a perfectly straight curve that runs between the joints FIGURE 1.15.

Figure 1.15.



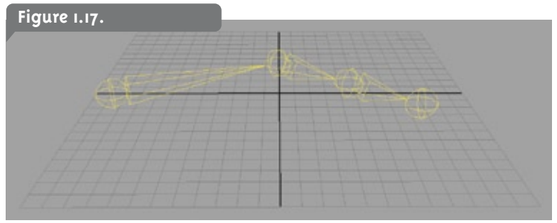
3. Activate the joint tool and hold down the 'C' key to snap the joint directly to the curve between the elbow and the wrist FIGURE 1.16.

Figure 1.16.



4. Now use the hypergraph to parent the forearm joint between the elbow and the wrist FIGURE 1.17.

Figure 1.17.

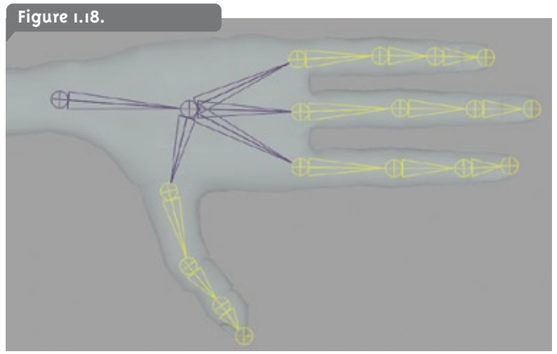


5. Delete the curve.

The placement of the palm joint is not overly important because it will not move. The palm joint is used to provide a way to plant the hand when using inverse kinematics. It will also act as the parent for all of the fingers.

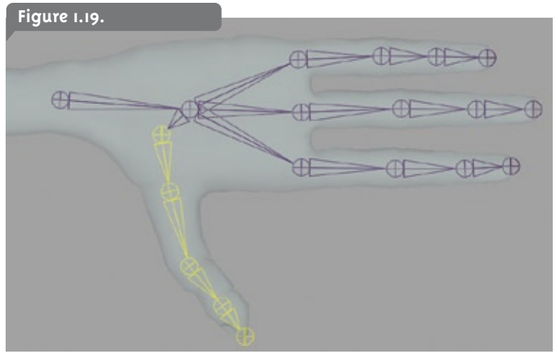
The finger joints are simple FK chains. A joint is placed at each knuckle and one at the tip to create a four-joint chain FIGURE 1.18. Regardless of whether these joints are to be controlled with FK or IK, you will need a pivot for each knuckle. Placement is easy, just try to get them as close as possible to where the underlying knuckle bone would be. I usually place the finger joints in a top viewport and then nudge them into position in the perspective view.

Figure 1.18.



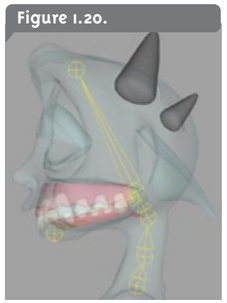
To create an opposable thumb, you can place another joint closer to the palm with which the entire thumb can rotate from FIGURE 1.19. This enables the creature to touch the pinky finger and the thumb together.

Figure 1.19.



The neck and head joints are fairly straightforward. Use an orthographic side viewport to place the two or more (depending on the length of the neck) neck joints and a single head joint. You can also place a jaw joint for use in facial animation FIGURE 1.20. The jaw joint is covered in detail in chapter 4.

Figure 1.20.



■ Finalizing Joints

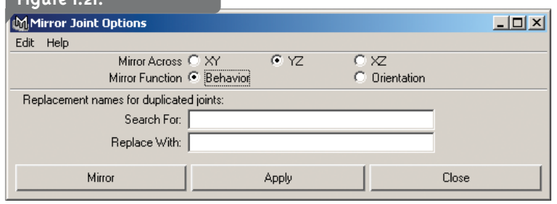
After moving all of your joints into position, you may find that some additional tweaking is necessary. To tweak the position of a joint without affecting it's children, you can hit the insert key to enter into pivot mode. Now move the joint and hit insert again to get out of pivot mode. Because a joint's position is defined by it's pivot, this will have the effect of moving the joint itself, without affecting the children.

After placing joints, it is a good idea to grab the root joint and freeze transformations on it. You may notice that a joint's translate channels are not zeroed after freezing transformations. This is normal. If you need to translate a joint, you can group it under a null to create clean translate channels. For a biped, the COG is the only joint that should be translated.

If the joints have been moved after they were created, their local rotation axes will be out of their default orientations. Re-orient the joints by selecting the root and executing `Skeleton > Orient Joints`. Use the default orient joint options.

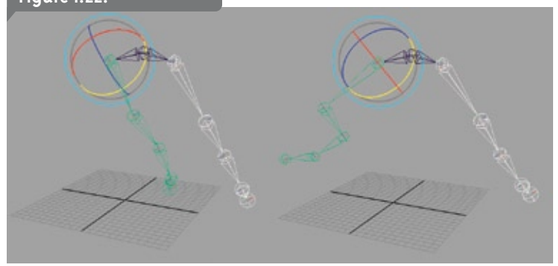
The default orientation ensures that the twisting axis is aligned properly but it does not ensure that the other axes are oriented correctly. This is especially important in areas of the body with hinge joints. Enter into component mode and click on the '?' icon. This will enable you to grab and tweak the local rotation axes from their defaults. Ensure that you do not rotate the local rotation axis such that the twist axis no longer points to the child. This is almost always unwanted.

Figure 1.21.



With all of the joints properly positioned, you can copy this hard work to the other side of the body using the `Skeleton > Mirror Joint` tool FIGURE 1.21. Notice that there are two ways of mirroring Maya joints, behavior and orientation. The orientation option will cause poses copied from one side, to the other, to look exactly the same. The behavior option is usually much more useful for creatures because it mirrors the behavior of the joint. In FIGURE 1.22 the left legs were mirrored with the 'orientation' option, the right ride was mirrored with the 'behavior' option. If a joint layout is mirrored using the behavior option, animation can be copied across the joints. This is especially useful for game creatures that have a lot of symmetrical cycle animations (walks, sneaks and runs).

Figure 1.22.



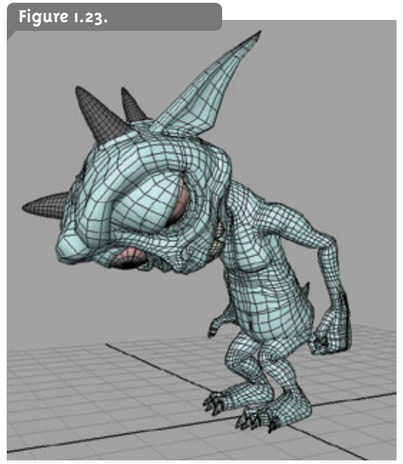
Finally, after you are sure that the orientations and positions are perfect, you can name the joints. Ensure that every joint in your hierarchy is given a meaningful and unique name. When you are ready to paint the smooth skin influence on a mesh, properly named joints will make your life much easier. A good naming convention is also necessary if you wish to use an auto-rigging script to automate any rigging processes. Make it a habit to name every node in your rig, not just the joints.

■ IK vs FK

Maya joints are animated, by default, in a manner known as FK or Forward Kinematics. Forward kinematics describes the behavior whereby children of a joint follow their parents. Motion in an FK joint chain moves down the chain or forward. This is often the desired behavior but it can present some problems in certain cases.

The easiest way to visualize the problem with FK is to consider a simple scenario where a character is standing still, and then must crouch down FIGURE 1.23. In this case, the leg joints must be animated so that as the COG joint is lowered (to crouch), the hip and knee joints rotate to keep the foot joints in the exact same spot (on the ground). If the hip and knee joints are not rotated perfectly, the foot will slid around thus breaking the illusion that they are firmly planted to the ground.

Figure 1.23.



This is where IK or Inverse Kinematics will save the day. With IK, the animator can specify a series of joints to be controlled by a single point, called an IK handle. Motion in an IK joint chain moves up the chain or inverted. If the hip and knee joints are controlled with an IK handle, they will automatically rotate themselves so that the end joint is in the exact same position as the IK handle. Thus, to plant a biped's feet, we place the IK handle on the ground and the legs will orient themselves to keep the feet planted.

Things get even more complicated if you take into consideration the fact that many scenes could benefit from having IK and FK (at different times). In these cases, the setup artist can build what is known as an IK/FK switch that allows the animator to specify exactly which mode they wish to use and when. Building an IK/FK switch is covered later in this chapter.

To create an IK handle, select `Skeleton > IK Handle Tool`. With the tool active, click on the start joint, then the end joint and press enter to create the IK handle. This handle can then animated to control the orientation of the joint chain.

There are three different types of IK solvers. Each one has a unique behavior and is useful for certain situations.

1. Rotate Plane (RP) IK Solver: This creates an IK chain with a pole vector. This pole vector describe the direction that the joint being solved should point in. You can constrain a pole vector on a RP IK handle to point in the direction of any object (like a locator). For a leg with an RP IK solver, this allows the animator to control the direction of the knee by moving an object that the knee points at.
2. Single Chain (SC) IK Solver: This is exactly like the RP IK solver except that there is no pole vector. You cannot control the orientation of the joints that are in an SC IK solver.
3. Spline IK Solver: This is a special type of IK solver that uses a NURBS curve to control the orientation of the joints. There are many uses for spline IK solvers. They are great for animating things like tentacles and antennae. In chapter 2, we will use a spline IK solver to animate the appendages on a creature.

The rest of this book will assume at least a basic understanding of the different IK solvers, and how IK behaves compared to an FK chain.

Iconic Representation:

Apart from designing believable deformations, a good rig must be easy to use. While some productions have their animators work directly with IK handles and joints, this can be problematic. If the creature is even remotely complex (like with fingers and toes), it can be extremely difficult to work with the joints. Joints can be difficult to distinguish in the viewport when they are surrounded by many other joints and a creature's mesh FIGURE 1.24. To make matters worse, they are all drawn exactly the same. If the animator wishes to see the creature's mesh while they work, they must continually move from wireframe/shaded to find and select the hidden joints. This creates an unfriendly workflow and is generally considered to be poor practice.

A far better approach is to use what is called iconic representation. With iconic representation, an animator never even touches the joints or IK handles themselves. Rather, an interface is created out of controller objects (usually NURBS curves or polygon objects), to control the rig

FIGURE 1.25. These controller objects are then animated, which in turn, animates the underlying joints.

Figure 1.24.

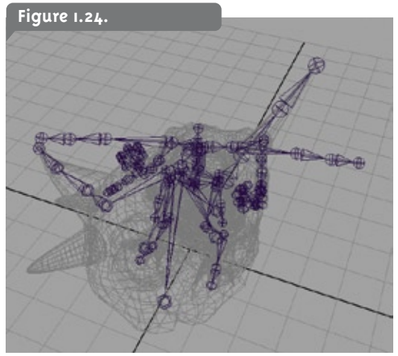
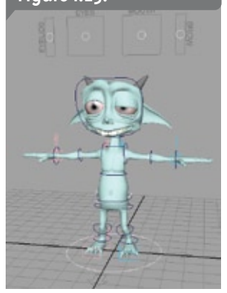


Figure 1.25.



Creating interfaces can be fun. Some setup artists are very particular about their setup styles and really take pride in designing easy-to-use controllers. Many controllers are designed to mimic their corresponding part of the body. A hand controller might be a curve in the shape of a hand. Similarly, a foot controller can be a foot shaped curve FIGURE 1.26. The important thing is that the controllers are clear, easy to find and select, and most importantly, they must affect the joints properly.

Figure 1.26.

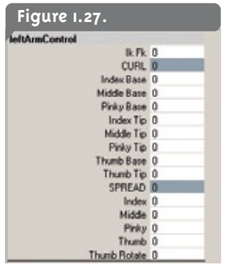
There are basically three different ways that a controller curve can affect a rig:

1. Orienting a Joint: The controller can be used to directly manipulate the orientation of an underlying joint. This is the most common type of controller as it is used throughout the body on most FK chains. This is usually done with an orient constraint.

2. Positioning a Joint or IK Handle: A controller object can also be used to control the position of a joint or IK handle. Usually this is only done on the root of a biped rig or for creating foot and hand IK controls. This is done by either point or parent constraining the joint or IK handle to the controller.

3. Housing Custom Attributes: Many functions of a control rig can be consolidated into one custom attribute. By adding a custom attribute to a controller curve, you can control many complex behaviors with a single value in the channel box FIGURE 1.27. Examples of this include curling fingers and toes. These sorts of attributes are usually connected to the underlying joints using more advanced connections like a MEL expression or set driven keys.

Figure 1.27.



There are many different ways to connect controller curves to your joint hierarchy. Usually, it is done with constraints, but you may wish to use direct connections, set driven keys, MEL expressions or utility nodes.

For the first type of controller object, there is a nice little trick you can use to ensure that the controller object's local rotation axes are aligned perfectly with the joint it is controlling (while maintaining zeroed transformations). This can be a tricky thing to do but it allows the animator to affect the controller curves in exactly the same way they would be affecting the joint. By having the controller's transformations zeroed, the animator can return to the default pose by entering a value of zero in the translate and rotate channels. To setup a controller object to control the orientation of a joint, do the following:

1. Create the controller object and position it near the joint as you wish.
2. Freeze the transformations on the controller.
3. Create a null group (`ctrl+g`).
4. Point snap the null to the joint.
5. Orient constrain the null to the joint. Now delete the orient constraint from the hypergraph. This will leave the null with the exact same local rotation axes as the joint.
6. Parent the controller object underneath the null.
7. Freeze the transformations on the controller object. Now the controller object should have perfectly zeroed transformations, but the local rotation axes are aligned with the joint.
8. To connect the joint, simply orient constrain it to the controller object.

If you choose to avoid this setup and just directly orient constrain the joint, the controller object will work fine but the animator may find it odd that the controller object's local rotation axes are aligned with the world (instead of the joint). Because it is impossible to affect the local rotation axes of a controller object without leaving garbage in the rotation channels, you must use the above method. This ensures that the controller object points in the correct direction and is zeroed. To have this controller object now affect the position of the joint, you can simply point constrain the joint.

■ Pick Walking

A common complaint amongst animators when using a rig with iconic representation is that of the inability to use pick walking. Pick walking is the term used to describe the method of moving through a hierarchy by using the up/down arrow keys. With regular Maya joints, pick walking is automatic. The animator can press the up arrow key to go from the wrist, to the elbow, to the shoulder and back down again. When you use iconic representation, pick walking is disabled and has no effect.

There are two ways of rectifying this (rather than just asking the animators to sacrifice their beloved workflow):

1. Use shape parenting to connect your controller curves rather than constraints.
2. Use a script to simulate pick walking.

The first method is rather complicated and involves disconnecting the shape node of the controller object and re-parenting it under the joint it is to control. In effect, when the animator selects a curve in the viewport, they actually select the joint (which is the new transform of the controller shape node). This way, the animator can still pick walk, because they are actually still animating on joints. Unfortunately, this method results in a controller object that does not have the ability to have it's color changed and it is somewhat of a pain to setup. Not only this, but the animator can break the rig by accidentally pick walking into a node that they were never intended to touch! Yuck!

The second method, that of using a script, is far better in my opinion because it allows not only regular up/down pick walking, but sideways pick-walking and explicit control over which control will be selected at each direction. The setup time is minimal and the controller objects can still have their colors changed. With a script, you get your cake and you can eat it too.

There is one script, in particular, that has gained relative fame for it's ability to handle pick walking on rigs with iconic representation. The jsPickWalk.mel script, written by Jason Schleifer, was created for use in the The Lord of The Rings movies to help Jason create the animation rigs for the film's main characters. Jason Schleifer is a well known name in the rigging community because of his excellent contributions including his many scripts and Alias training DVDs.

To use his pick walking script, copy the script from the DVD to your scripts directory. Setting up pick walking involves two steps.

1. Execute the following MEL command for each pick walk direction on each controller. Replace 'down' with whatever direction you want (up/down/left/right).

```
jsMakePickWalk controllerObjectName targetObject down;
```

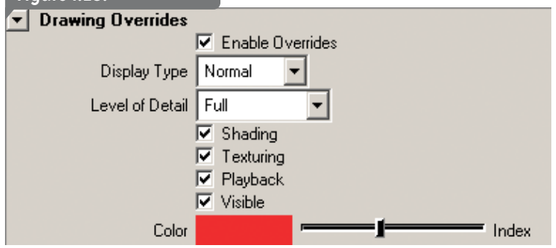
2. Now to make the pick walking actually work, you must edit your Maya hotkeys to execute the script's pick walk procedure. The procedure is simply 'jsPickWalk down;'. You can either edit the default pick walk keys, or make this a special type of pick walking that is done with the shift+direction arrow key. Either way, the animators will appreciate having this kind of flexibility.

■ Pretty Colors

Once all of the controller objects are connected and working, you can add some color to further help differentiate the viewport clutter. As a general rule, controller objects on the left hand side of the creature should be colored blue. Controllers on the right, should be red. These coloring conventions are borrowed from aviation and sailing where buoys are traditionally colored red on the right and blue on the left. These colors are generally accepted as being standard throughout the industry. Controllers in the middle of the body can be any color, but white or yellow stand out best.

To edit the color of a controller object, open the attribute editor and find the 'Object Display' tab. Open this and scroll down to the 'Drawing Overrides' section. Opening this will reveal some grayed out controls. Check the 'Enable Overrides' button to use the color swatch FIGURE 1.28.

Figure 1.28.



Adding colors to your rig only takes a few minutes but can really help a rig feel easier to use. The animator can intuitively tell which side of the body he is working on, regardless of where the camera is pointing.

Cleaning Your Creature:

Before you send a creature off to the animators, you must ensure that it is clean, robust and bulletproof. This means that the animator should not be able to accidentally break the rig. This can be done if the animator is given access to an attribute that should not be affected. In addition, the rig file should contain no extra nodes of any kind. Depending on the type of production, you may also wish to organize the DAG nodes of your creature in the hypergraph. This will allow for easy importing into sets and layouts. This section includes some tips and tricks to keep your rigs in good working order.

■ Dirty Channels

The problem of dirty channels must always be addressed with Maya rigs. The problem is that the rig is likely controlled entirely by transform nodes. Whether these are controller curves, polygon objects or joints, they are all created, by default, with translate, rotate and scale attributes FIGURE 1.29. For many cases, the controller is likely only meant to affect the orientation of a joint. It is very rare that translate and especially scale attributes should be edited by the animator. Doing so will usually screw-up the rig and cause a great mess.

Avoiding this is simple. Click and drag over all of the attributes in the channel box that are not needed. You can make multiple selections by holding down the `ctrl` key. With the unwanted channels selected, right click on them and choose 'Lock and Hide Selected' FIGURE 1.30. This will

lock the attributes, preventing them from being changed, and make them non-keyable by hiding them from the user.

Figure 1.29.

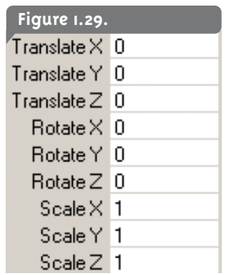
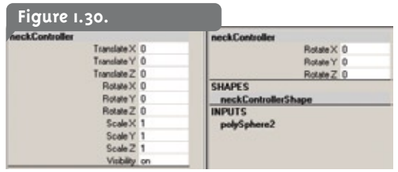
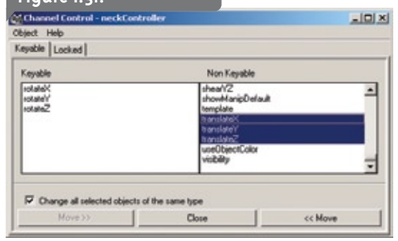


Figure 1.30.



You can use `Window > General Editors > Channel Control` to bring up an interface for restoring locked and hidden attributes FIGURE 1.31. This may be necessary if you accidentally hide an attribute.

Figure 1.31.



Be mindful of exactly what attributes are showing in the channel box. Leaving unwanted attributes is one of the easiest ways to create a breakable rig.

■ Hypergraph Organization

Keeping your creature's DAG nodes in order is extremely important for the well being of your pipeline. This means that absolutely no unused nodes should be present in the creature scene file. This is why it is important to name everything in the scene. With every node properly named, you will be able to determine whether or not it belongs.

While it may seem anal to name and track every node in your scene, the importance of doing so becomes glaringly apparent once you begin to import your creatures into their animation scenes. Loose, unused nodes can quickly pile-up creating a very disorganized and chaotic mess. Then, if you need to remove a creature from a scene, it can be difficult to track every node that is associated with it for proper deletion.

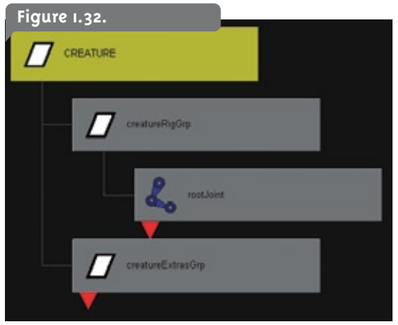
Not only must every node in the creature file have a purpose, but it should be organized into it's proper parent group. A creature will require at least two main groups, they are:

1. Creature Extras Group: This group should contain all of the skinned geometry, the low-res geometry and the iconic representation transform nodes. Many people simply parent their iconic representation controllers directly into the joint hierarchy. I would not recommend this. Doing so can create an unnecessarily convoluted rig hierarchy that is difficult to debug. Since version 5.0, Maya has included a parent constraint. This can help you connect your controller objects to the rig without having to make the rig hierarchy bloated with extra nodes. As you will undoubtedly need to dig around in the joint hierarchy, having it clean and object free will make your life much easier. The extras group node should also contain any facial animation interfaces and extra dynamic or deformation nodes. Have each extra part of the rig grouped into it's own category and placed in this main group node.

2. Creature Rig Group: This group contains the actual rig itself. In order to keep it clean and easy to read, this hierarchy should contain only joints and their constraints. If you choose to parent controller objects directly into the rig group, it may become difficult to work with. It is far better to to keep this hierarchy clean from extra nodes. Never parent the mesh geometry into this group. If a mesh is parented into a hierarchy of joints that it is bound to, it will create what is known as a double transformation. These occur when an object has a transformation on both it's tranform and shape nodes. The mesh will fly off into space as it is affected by both the skincluster and the parenting. The rig group contains the lowest-level pieces of the rig and should be separated from the high-level control objects and any mesh geometry.

With everything nicely organized into their respective main groups, you can then group both of these main groups together to create one single node that contains the entire creature FIGURE 1.32. This will make identifying the creature much easier when it is imported into an animation scene that may contain set pieces and other creatures.

Figure 1.32.



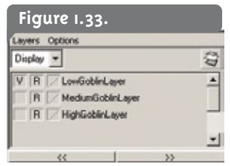
Beginner riggers always make the mistake of never using the hypergraph. This inevitably creates a rig file is chalk full of extra garbage nodes and poorly organized. If you want to create a creature that will flow through a production while being easy to edit and maintain, you must be diligent about organizing the DAG. In the long run, proper organization can make or break a production. Do not underestimate the importance of keeping your creature files clean.

■ Layers and Multi-Resolution Rigs

Maya provides an excellent tool for keeping the visibility of a creature's nodes organized. The layer editor is extremely useful for creature rigs. The most common way to organize the visibility of a creature is by it's resolution.

Film and television pipelines often utilize a multi-resolution creature rig. This means that the creature has various different resolutions with which it can be viewed. Usually there are three main resolutions, although two can work just fine FIGURE 1.33.

Figure 1.33.



Low-Res Layer- This layer contains polygon geometry that is parent constrained to the joints. It is easy to create a low-res rig. Use the polygon extract tool to slice up the mesh into pieces for each joint. This creates a rig that can be animated in real time. Higher resolution meshes can become quite slow when deformations are applied. A low-res rig provides a quick way of animating the character without needing to calculate the slower deformation rig.

Medium-Res Layer- This layer contains the geometry that is smooth bound to the skeleton. Some productions may find it useful to create a medium resolution rig that can be used for playblasts and dailies. A medium resolution rig is often too slow to animate with, but it can give the animator a better idea of exactly how their poses are going to look. The medium resolution layer contains the facial rig, if the low-res does not.

High-Res Layer- Some film productions may choose to have a layer with the high or render quality mesh. This is either a highly smoothed polygon mesh or a subdivision surface. This is the slowest layer and may contain many other deformers or even a complete muscle simulation. You may even choose to keep put this rig in a completely separate file, rather than just a layer. This can prevent the animator from accidentally destroying the deformation setup while keeping the animation file more compact. No animation is ever done on the high resolution rig. Many productions prefer to copy the animation from the low-res rig into a completely separate file that contains the render-ready rig. This is called animation transfer and is covered in chapter five.

Even if you have turned the main rig group's visibility to 'off', you can further prevent the animator from ever accidentally touching the rig by throwing it into it's own layer and turning the visibility of the entire layer 'off'. This optional layer should be named something like 'DO NOT TOUCH', to ensure that the animator does not get curious.

Figure 1.34.

Layers are an excellent tool for creating multi-resolution creature rigs. Do not ignore them! Finally, you can turn the visibility of everything to 'off' in every viewport panel FIGURE 1.34. Then turn 'on' only the objects that the animator should see, usually NURBS curves and polygons.

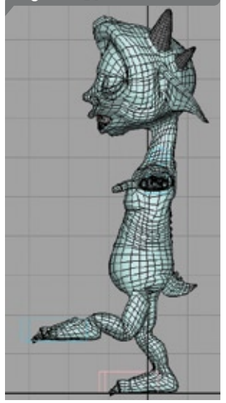
Following these fail safes will result in a rig file that is clean, efficient, easy to use and difficult to break. Creating clean rigs will reduce the stress throughout the studio and will make the production much smoother.

The No-Flip IK Knee:

Aside from the logistical tasks of creating a good creature rig, you must also consider the needs and wants of the animators. This means finding ways to make the rig easier to control and faster to animate. One of the age-old complaints about IK chains is that of the pole vector constraint flipping. Recall that the pole vector is constrained to an object so that (in the case of a leg), the knee will point in the direction of the controller object.

This provides a nice way to control the direction of the knee, but it can create a some headaches for the animator. The problem arises when the leg must be animated past the current position that the pole vector is in. This causes the leg chain to flip backwards, breaking the knee FIGURE 1.35 . To avoid this, the animator must meticulously place the knee controller for each pose such that it prevents the knee from flipping.

Figure 1.35.



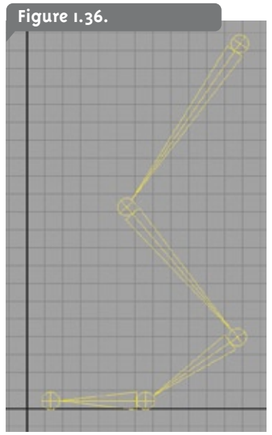
To avoid this problem, the knee can be setup using a different technique. This setup, called the 'No-flip Knee', will create an IK leg with a knee that automatically points in the direction of the foot. Additionally, it provides a way for the animator to still control the direction of the knee explicitly. Many 'automatic' setups take control away from the animator. This setup does not.

Exercise 1.1 **The No-Flip Knee**

To start off with, this exercise will demonstrate how to setup a no-flip knee in an empty scene. At the end, you will be show how to do it on a character who's leg joints are not axially aligned.

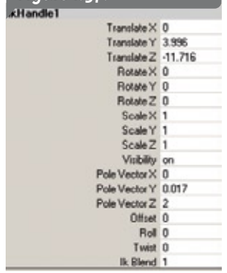
1. Drop into a side orthographic viewport and create the leg joints like in FIGURE 1.36. Start from the hip and create a knee, ankle, ball and toe. To ensure that the toe joint is perfectly flat, hold the `shift` key while placing it.

Figure 1.36.



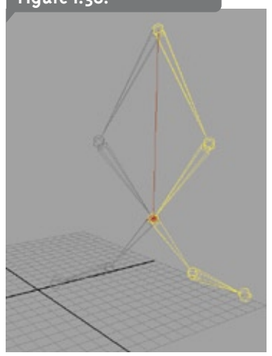
2. Open exercise1.1_Start.mb to see the joints after being placed. Create a rotate plane IK handle from the hip to the ankle (`Skeleton>Ik Handle Tool`). This will result in an IK handle with the following attributes in the channel box FIGURE 1.37.

Figure 1.37.



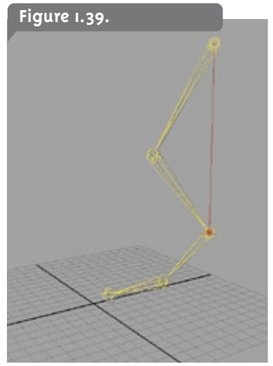
3. Select the IK handle and open the channel box. Edit the 'Pole Vector X' value to read 0.1. Set the 'Pole Vector Y' and 'Pole Vector Z' to 0. This will cause the leg to twist 90 degrees FIGURE 1.38.

Figure 1.38.



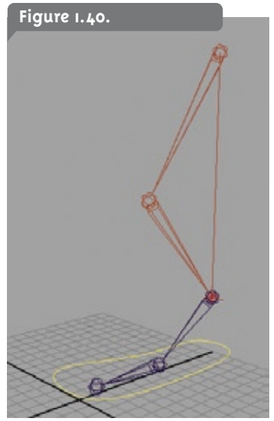
4. To point the leg straight again, enter a value of 90 into the twist attribute on the IK handle FIGURE 1.39.

Figure 1.39.



5. To control the leg, we will use a NURBS curve. Create a NURBS circle (`Create> NURBS Primitives> Circle`). Snap this circle to the ball joint and position the control vertices to look roughly like a foot FIGURE 1.40. Name this curve 'footControl'.

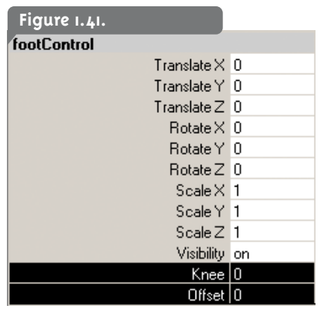
Figure 1.40.



6. Freeze the transformations on the controller object.

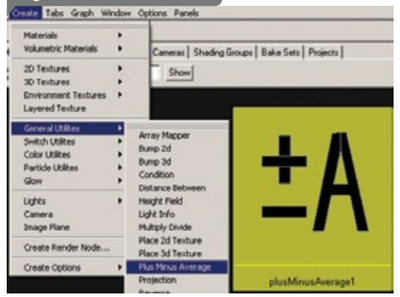
7. Select the foot control curve and choose `Modify>Add Attribute`. Add two float attributes to the control curve named 'knee' and 'offset'. The offset attribute will be used to straighten the leg, the 'knee' attribute will eventually control the direction of the knee FIGURE 1.41.

Figure 1.41.



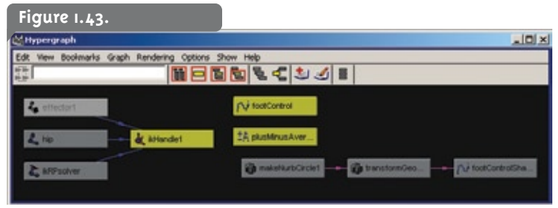
8. Now open the hypershade and choose `Create>General Utilities>Plus Minus Average`. This will create a utility node FIGURE 1.42. This node can be plugged into the dependency graph to add, subtract, or average two inputs and then send the result to downstream nodes.

Figure 1.42.



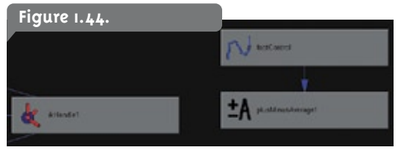
9. Open exercise1.1_MiddleB.mb to see the setup thus far. Shift select the IK handle, plusMinusAverage node, and the foot control curve. With these three objects selected, open the hypergraph and hit the 'Input/Output Connections' button. This will display the dependency graph for all three nodes FIGURE 1.43.

Figure 1.43.



10. We are going to use the hypergraph to edit some dependency graph connections. To start with, right-click on the right hand side of the foot control node and choose the knee attribute from the pop-up menu. With the knee attribute selected, right-click on the left side (input side), of the utility node (named plusMinusAverage1). This will bring up another floating menu. Choose the 'input1D[0]' input attribute. This will make a connection in the dependency graph that is drawn as a blue line connecting the foot control node to the plusMinusAverage1 node FIGURE 1.44.

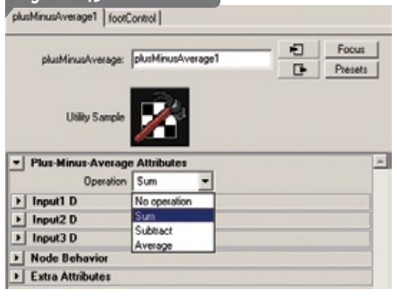
Figure 1.44.



11. Make another connection by plugging the offset attribute from the foot control node into the 'input1D[1]' attribute of the plusMinusAverage1 node.

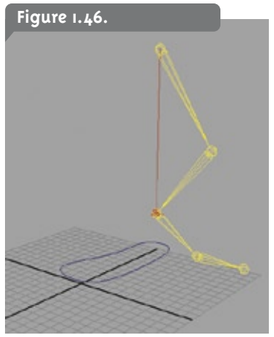
12. Now the plusMinusAverage1 utility node is adding these inputs together. This is the default behavior for this type of utility node. You could open the attribute editor to edit the operation (change to minus or average), but we want the node to sum the inputs FIGURE 1.45.

Figure 1.45.



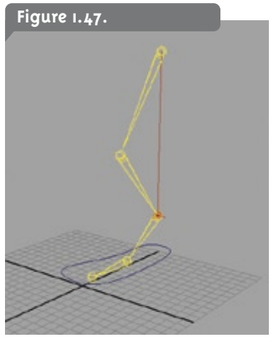
13. Now connect the output1D from the plusMinusAverage1 node to the 'twist' attribute on the IK handle node. This will cause the leg to twist 90 degrees in the viewport FIGURE 1.46.

Figure 1.46.



14. Open exercise1.1_MiddleC.mb to see the finished connections. To correct the twisting, enter a value of 90 into the offset attribute on foot control. Now the leg is pointed straight ahead FIGURE 1.47.

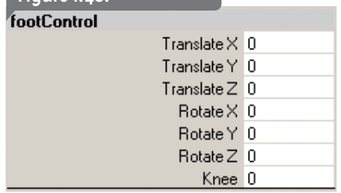
Figure 1.47.



15. To finish up the controller object, select the scale, visibility and offset attributes in the channel box. With the unwanted attributes selected, right-click in the channel box and choose 'Lock

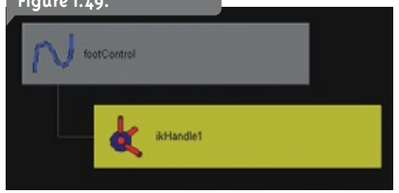
Selected'. This will clean up the channel box FIGURE 1.48.

Figure 1.48.



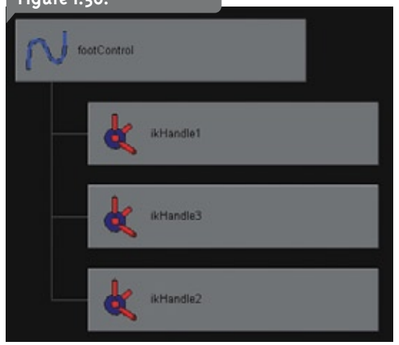
16. Now select the IK handle, shift select the foot control and hit the `p` key. This will parent the IK handle under the foot control FIGURE 1.49. The no-flip setup is complete, but the foot control could be stabilized by controlling the foot joints with inverse kinematics.

Figure 1.49.



17. Create a single chain IK handle from the ankle to the ball and from the ball to the toe. Parent these two IK handles under the foot control curve FIGURE 1.50.

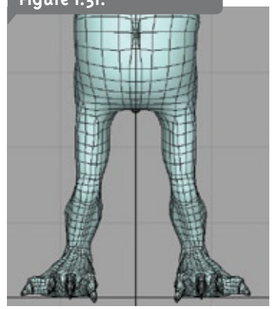
Figure 1.50.



18. Open exercise1.1_Finished.mb to see the completed leg setup. Notice that after adding the two new IK handles to the foot, the foot remains flat regardless of where the foot is moved. Rotating the foot control orients the entire leg, including the knee. Also notice that regardless of where the foot is positioned, the knee will not flip.

The setup procedure presented above is perfectly fine if you creature's leg has been modeled in a perfectly straight manner. But what if the leg is in a more relaxed pose? In the case of the Goblin creature in this chapter, his legs are modeled in a slightly bent out fashion FIGURE 1.51. This makes it impossible to use an orthographic joint chain. The no-flip knee setup can be setup on a creature like this, but it takes some extra steps.

Figure 1.51.



If you are familiar with the setup, you will recall that it requires that you enter an offset value to position the leg back into it's default pose. For a perfectly flat leg chain, we can enter an offset of 90. For the Goblin, I had to arrive at a different offset value. So how does one find the proper offset value for a non-straight leg chain?

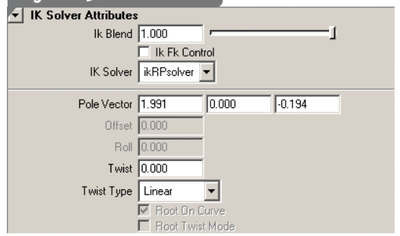
It is actually quite simple. Duplicate the leg chain *before* you begin the setup. Throw this duplicate into a separate layer and template this layer. This will act as a guide. At setup number 13 (in the above exercise), instead of entering a value of 90, you can middle mouse drag on the offset attribute until the leg chain lies directly on top of the duplicate. Remember, the duplicate is in the default pose, so when the leg chain lines up with the duplicate, it is in it's default pose. This is the offset that you need to type in. This works just fine and will allow you to create this exact setup without the need to do it on an orthographic joint chain and then group and reposition. This is a pain to do properly and causes the leg chain to contain garbage default transformation values after un-grouping.

The Classic IK/FK Switch :

It is not uncommon that a scene will require both IK and FK. In order to create a rig that is capable of using both, we can create what is known as an IK/FK switch. This allows the animator to control the limb using either forward or inverse kinematics and they can even blend between each mode.

IK/FK switches are considered standard throughout the industry. Animators these days are spoiled, they will need IK/FK switches in all but the most simple rigs. Recognizing the importance of such a feature, Alias decided to include a sort of automatic IK/FK switch on all IK handles since Maya version 5.0 FIGURE 1.52. Unfortunately, Maya's native IK/FK switch is somewhat problematic and for all intents and purposes, it is inferior to a handmade one.

Figure 1.52.



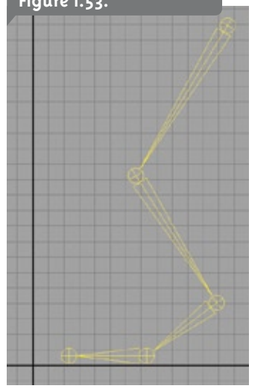
It may take slightly more time to create a classic IK/FK switch, but the results are well worth it. The basic concept involves using three separate joint chains. One joint chain is the actual joints that the mesh is bound to. The second chain is posed using regular forward kinematics. The third chain is controlled with an IK handle. By orient constraining the 'real' joint chain to these 'fake' IK and FK versions, we can have the rig controlled by either. To blend between IK and FK, we blend the orient constraint's weight value. This creates the switch. The easiest way to understand an IK/FK switch is to create one from scratch, so let's do it!

Exercise 1.2 **Creating a Classic IK/FK Switch**

This setup will create an IK/FK switch on an arbitrary leg. Please understand that this concept can (and should) be used for both legs and arms.

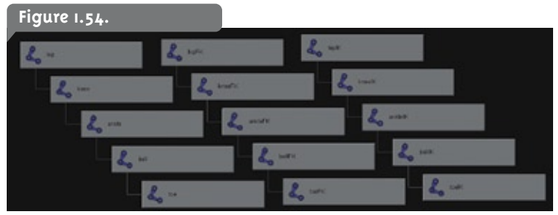
1. Open a new scene file and create a hip, knee, ankle, ball and toe in this configuration FIGURE 1.53.

Figure 1.53.



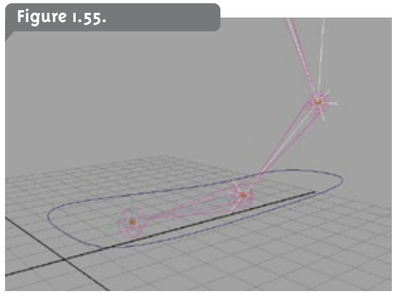
2. Open exercise1.2_Start.mb to see the joint layout. Select the hip joint and choose `Edit>Duplicate` (with default settings). Duplicate the entire joint chain twice. This will leave you will three separate chains. Tack the extension 'IK' and 'FK' onto the end of the names of the newly created joint chains FIGURE 1.54 .

Figure 1.54.



3. Open exercise1.2_MiddleA.mb and look in the hypergraph to see the new joint chains. We are going to setup a very simple IK leg to show how the blending works. To do this, hide the FK and real joint chains (you can turn the visibility of the root joints to 'off'). Now create three single chain IK handles from the hipIK to the ankleIK, ankleIK to ballIK and ballIK to toeIK FIGURE 1.55.

Figure 1.55.

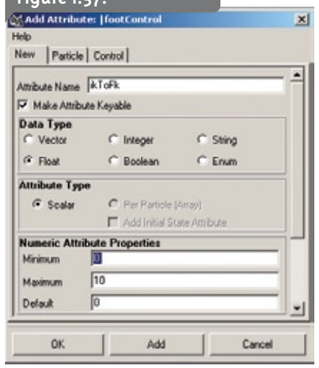


4. Create a NURBS curve circle (`Create>NURBS Primitives> Circle`). Point snap it to the ball joint, scale and position the curve to look like a foot. Freeze the transformations on this curve and rename it 'footControl'. Parent the IK handles under this control to finish the IK leg setup FIGURE 1.56. Please note, this is a very simple IK leg setup and is just for demonstration.

Figure 1.56.

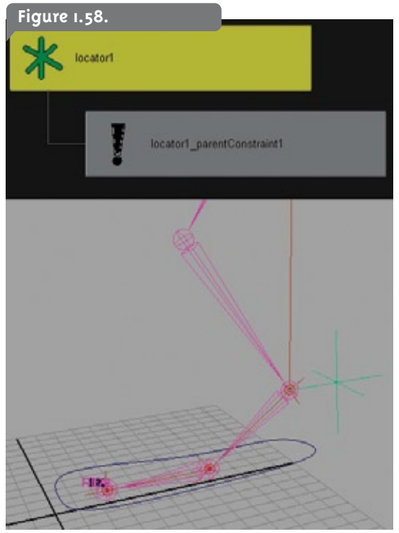
5. Open exercise1.2_MiddleB.mb to see the finished simple IK leg setup. Turn the visibility of the FK and real joint chains back to 'on'. We are now ready to build the switch. Choose `Create > Locator` to make a locator. Select this locator and name it 'IKtoFK'. Add a custom float attribute to the locator called 'IKtoFK'. In the add attribute options, set the minimum to 0 and the maximum to 10 FIGURE 1.57.

Figure 1.57.



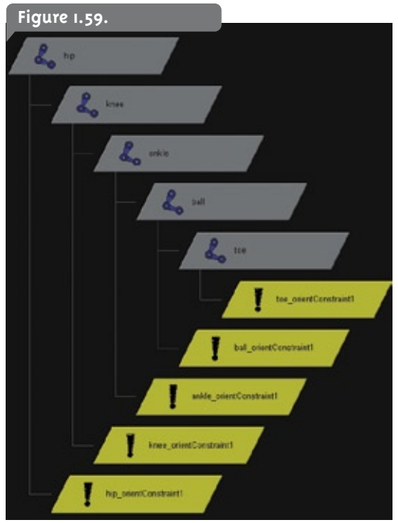
6. Move the locator near the ankle joint and then parent constrain it to the ankle FIGURE 1.58. This locator will act as the switch so it should follow the real joint chain, regardless of whether it is in FK or IK.

Figure 1.58.



7. So that the real joint chain will follow the IK and FK copies, we must orient constrain each real joint to it's corresponding IK and FK duplicate. Select each joint, shift select it's corresponding joint and choose `Constrain > Orient`. Do this for each joint, constraining to both the IK and FK versions FIGURE 1.59.

Figure 1.59.



8. The switch is almost finished. To tie everything up, we must connect the weight of each of the orient constraints to the custom attribute on the controller curve. You can do this with set driven keys or an expression. It really does not matter. When the 'IKtoFK' attribute is equal to 0, the constraints to the IK leg should all be equal to 1, the FK 0. When the 'IKtoFK' attribute is equal to 10, the constraints to the FK leg should equal 1, the IK 0. This way, the leg will blend between the IK and FK copies as the custom attribute blends from 0 to 10.

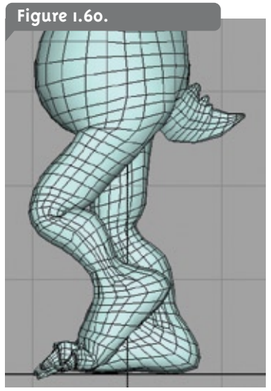
9. Open exercise1.2_Finished.mb to see the final IK/FK switch. Try moving the foot control and then blending the custom attribute on the locator to see the leg blend.

This setup can be done for both legs and arms. In addition, this setup will work with a reverse foot or an auto-forearm.

Creating a Group-Based Reverse Foot :

A 'reverse foot' is the name given to an IK foot setup that allows the animator to control the foot as though it were planted on the ground. This allows motions like peeling the heel off the ground (while keeping the toes planted) FIGURE 1.60.

Figure 1.60.



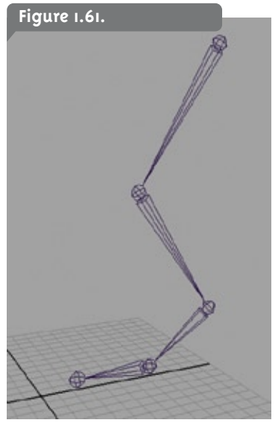
While many books and tutorials teach a reverse foot that is done with a separate 'reverse' joint chain, this setup is a little bit more robust. Unlike the joint-based reverse foot method, this allows full control over tapping the toe. In addition, there are no extra joints in the scene creating a cleaner looking rig file.

Exercise 1.3 Rigging a Group-Based Reverse Foot

This setup is applied on top of an IK leg. A reverse foot setup is only possible on an IK leg chain because the IK handles provide a method of reversing the motion. This exercise is an excellent introduction to creating hierarchies of control. If you understand how this setup works, you will be able to use the same concepts in many other situations.

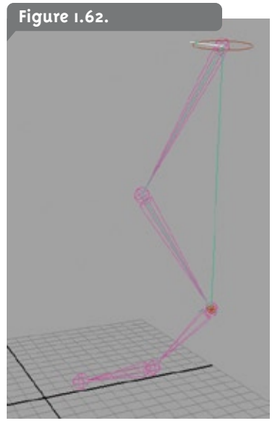
1. Open exercise1.3_Start.mb. This scene file contains a simple joint chain, like that used in a leg. There is a hip, knee, ankle, ball, and toe joint. These are all currently FK joints FIGURE 1.61.

Figure 1.61.



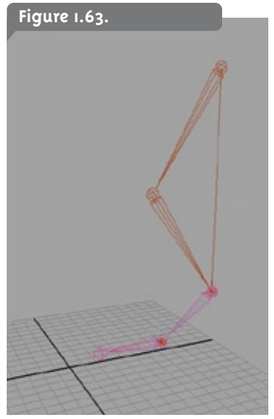
2. Create a rotate plane IK handle from the hip to the ankle FIGURE 1.62. Name this IK handle 'ankleIK'.

Figure 1.62.



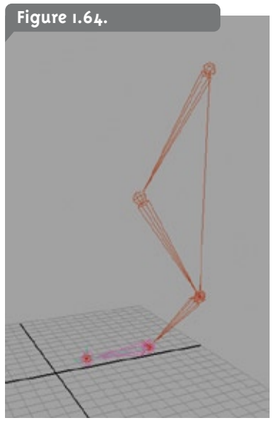
3. Create a single chain IK handle from the ankle to the ball FIGURE 1.63. Name this IK handle 'ballIK'.

Figure 1.63.



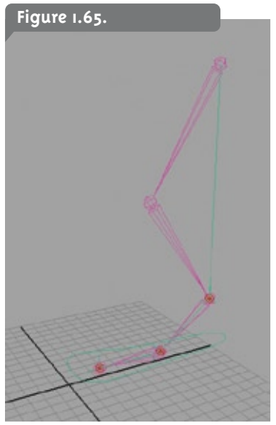
4. Create another single chain IK handle from the ball to the toe FIGURE 1.64. Name this IK handle 'toeIK'.

Figure 1.64.



5. Open exercise1.3_MiddleA.mb to see the leg setup with IK. Now we must create a controller that can be used to animate the entire leg. This control object will be able to translate and rotate the foot, as well as house some custom attributes that will later control the reverse foot actions. Create a NURBS circle (`Create > NURBS Primitives > Circle`) and name it 'footControl'. Point snap this to the ball joint and then scale it out. You can manipulate the control points to shape the circle like a foot FIGURE 1.65.

Figure 1.65.

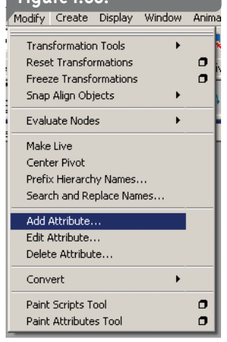


6. Open exercise1.3_MiddleB.mb to see the finished foot controller curve. With the foot control curve selected, choose `Modify > Add Attribute` FIGURE 1.66. We need to add several different custom float attributes that will later be used to control the reverse foot. Add the following attributes to the foot controller:

peelHeel: This will control the action that peels the heel of the floor while keeping the toes planted. Make this a float with a minimum value of 0,

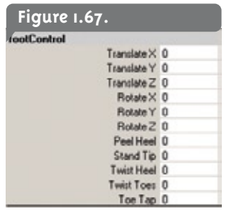
and a maximum value of 10.
standTip: This will control the action of standing the foot on it's toes. This is a float from 0 to 10.
twistHeel: This action twists the entire foot from the heel. Twist heel is a float from -10 to 10.
twistToes: This action twists the entire foot from the toes. Twist toes is a float from -10 to 10.
toeTap: This action will tap the toes up/down. The toe tap attribute is a float from -10 to 10.

Figure 1.66.



7. Open exercise1.3_MiddleC.mb to see the foot controller with the custom attributes. To polish the controller off, select the scale and visibility attributes in the channel box and right-click to bring up the channel box menu. Choose 'Lock and Hide Selected' to lock these attributes and hide them from the animator FIGURE 1.67.

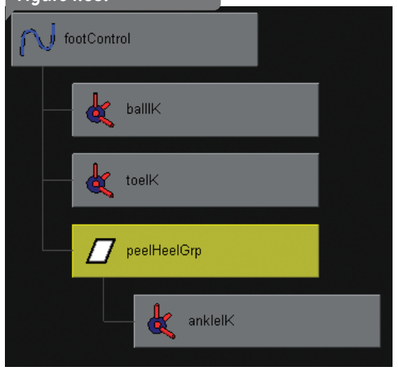
Figure 1.67.



8. Now we are ready to start the reverse foot setup. This setup works by grouping the IK handles under null group nodes. These null groups can have their pivot points placed wherever we wish. To control the foot, we simply rotate these group nodes to move the IK handles which in turn affect the joints. It may easiest to open up the hypergraph to see what you are doing while you setup a reverse foot.

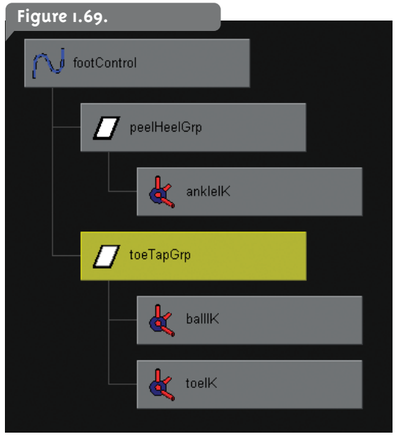
9. Start by selecting the ankleIK and hitting `ctrl+g`. This will group the IK handle to itself. With the newly created group node selected, activate the move tool, then press the `insert` key to go into pivot mode. Snap the pivot of this group node to the ball joint on the skeleton. Hit `insert` to exit out of pivot mode. Name this first group 'peelHeelGrp' FIGURE 1.68. This is the pivot point from which the heel will peel off the ground.

Figure 1.68.



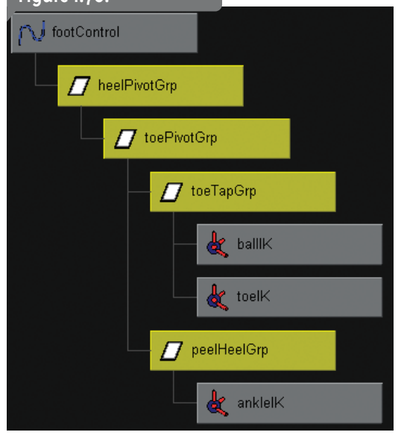
10. Open exercise1.3_MiddleD.mb to see the peel heel group with it's properly positioned pivot point. If you select the group node and rotate it in the X axis, you will see the heel peeling motion that we are going to connect via set driven key to the foot control curve. To continue the setup, select the toe and ball IK handles. Hit `ctrl+g` again to put these under a new group node. Enter into pivot mode and snap the pivot of this group to the ball joint as well. Name this new group, 'toeTapGrp' FIGURE 1.69.

Figure 1.69.



11. Open exercise1.3_MiddleE.mb to see the toe tapping group properly setup. Now select the toeTapGrp and peelHeelGrp and hit `ctrl+g` to group these under a third group node. Name this group 'toePivotGrp' and snap the pivot point to the toe joint. To finish the group-based hierarchy, select the toePivotGrp and group it again. Name this new group 'heelPivotGrp'. Snap the pivot of the heel pivot to the ankle joint. This completes our reverse foot grouping FIGURE 1.70.

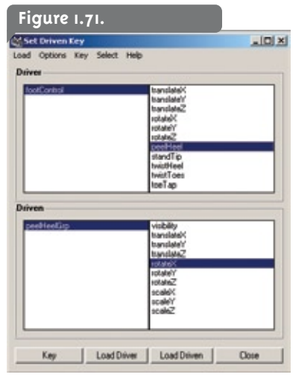
Figure 1.70.



12. Open exercise1.3_MiddleF.mb to see the finished reverse foot group hierarchy. While we have built all of the control necessary to pose the foot, it would currently require that the animator use the hypergraph to find the necessary group nodes. This would be a horrible animation workflow. To speed things up, we will connect the rotation of the foot groups to the custom attributes on the foot control curve. This will enable the animator to quickly and easily pose the foot.

13. Start by opening the hypergraph and locating the foot hierarchy. You will need to select them easily while setting up the driven keys. `Choose Animate > Set Driven Key > Set` and open the option box to see the set driven key window. Load the foot control curve as the driver and select the peel heel attribute as the driver attribute. Load the peelHeelGrp as the driven object and select the rotateX attribute as the driven value. With both attributes at 0, set a key FIGURE 1.71. Then select foot control curve and set the peel heel value of 10 (maximum). Now grab the peelHeelGrp node and rotate in X to about positive 45 degrees. Set another key. This completes the peel heel motion. Try middle-mouse dragging on the peel heel attribute to see it work.

Figure 1.71.



14. Open exercise1.3_MiddleG.mb to see the setup with the completed peel heel function. Let's move up the hierarchy and setup the toeTapGrp. This group is driven by the toe tap attribute on the controller curve. Load the toeTapGrp into the driven section of the set driven key window and set a key at 0,0. Set the driver attribute to -10, set the toeTapGrp's rotateX to 74 and set another key. Make the final key with the driver at 10 and the group's rotateX at -45. This completes the toe tap function.

15. Open exercise1.3_MiddleH.mb to see the setup with the finished toe tap action. Let's move on and quickly set the rest of the keys:

Stand Tip Driver Value	toePivotGrp Driven Value
0	RotateX = 0
10	RotateX = 40

Twist Heel Driver Value	heelPivotGrp Driven Value
0	RotateX = 0
-10	RotateX = -50
10	RotateX = 50

Twist Toe Driver Value	toePivotGrp Driven Value
0	RotateX = 0
-10	RotateX = 50
10	RotateX = -45

16. Open exercise1.3_Finished.mb to see the setup with the finished set driven keys.

The reverse foot setup is a great help for animators. To make an even better leg setup, you can do a reverse foot setup on top of a no-flip IK knee and an IK/FK switch. All of these setups can be combined to create a very nice leg setup with plenty of control for almost any situation.

The Auto-Forearm Twist:

The forearm is a tricky part of the body to rig. Many novice riggers will make the mistake of using only an elbow and a wrist joint. While this can work, it will require that you setup an influence object to simulate the twisting motion of the forearm. This is more bother than it is worth and can be completely avoided by using an intermediary forearm joint.

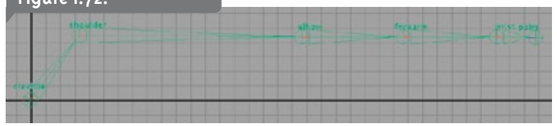
The way this setup works is by using a controller curve to control both the wrist and the forearm joints. It is quite simple, but it must be setup carefully to avoid problems.

Exercise 1.4 **Auto-Forearm Twisting**

1. Open exercise1.4_Start.mb to see the joint chain we will be working on. This chain includes a clavicle, shoulder, elbow, forearm, wrist and palm

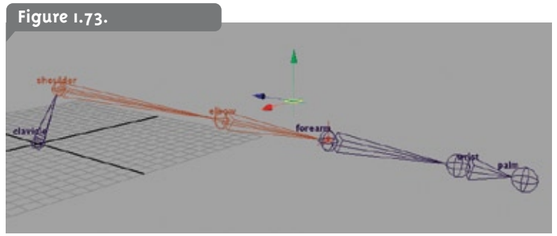
FIGURE 1.72. While this setup will work fine on an FK or IK chain, it will be easier for us to test the setup with an IK handle. Create a rotate plane IK handle from the shoulder to the forearm. Do not make the IK handle to the wrist, this would cause the forearm joint to bend (thus breaking the arm). Name this IK handle 'armIK'.

Figure 1.72.



2. To control the direction of the elbow, we will constrain the rotate plane IK handle's pole vector to a locator. Create a locator (`Create > Locator`) and point snap it to the elbow joint. Now move this locator back in the viewport behind the elbow FIGURE 1.73. Name the locator 'elbowControl'.

Figure 1.73.



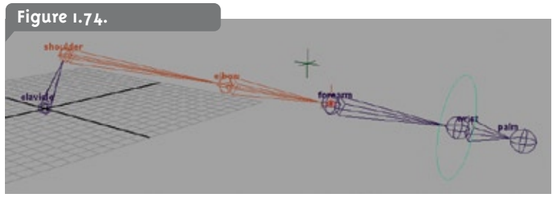
3. Freeze the transformations on the locator. Now select the locator, shift select the IK handle and choose `Constrain > Pole Vector`. The IK handle controls the arm joints, and the locator now controls the direction that the elbow will point in.

4. Open exercise1.4_MiddleA.mb to see the IK setup thus far. While the IK handle will certainly work fine from where it is, we want to combine all of the arm control into one single controller curve. This controller curve will be located at the wrist and in addition to twisting the forearm, this curve will be a parent of the IK handle. This will allow the animator to pose the arm and twist the wrist with one single controller. We will take care of this problem after setting up the twisting control.

5. Create a NURBS circle to use as a wrist controller (`Create > NURBS Primitives > Circle`). Now point snap this circle to the wrist joint. Rename this curve 'armControl'. You can scale this curve up slightly to make it easier to select in the viewport. Enter a value of -90 into the curve's rotateX. This will line it up perpendicular to the joint FIGURE 1.74.

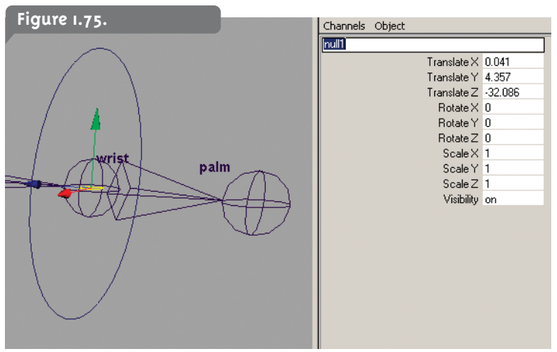
6. Open exercise1.4_MiddleB.mb to see the controller curve scaled and positioned properly. This curve will be connected to the rotation of both the wrist joint and the forearm joint. In order to simulate the interaction between the radius and the ulna skeleton, the forearm joint will handle all of the twisting motion (X axis). The wrist joint will only rotate up/down and left/right. These are the Z and Y axes respectively.

Figure 1.74.



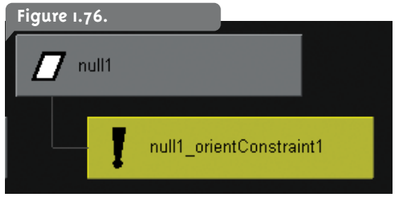
7. To setup the twisting, start by freezing the transformations on the control curve. This will zero out the transformations. The problem now is that the local rotation axis on the control curve is different than the joint. Lining these up will create a more stable control. To do so, clear your selection and press `ctrl+g`. This will create an empty group node. Now point snap the group to the wrist joint. Rename 'null1', 'wristGrp' FIGURE 1.75.

Figure 1.75.



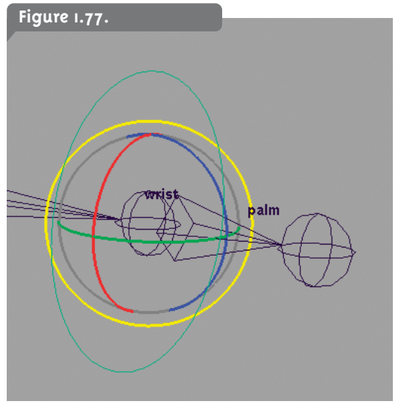
8. You may need to open the hypergraph or outliner to select the newly created group node. Select the wrist joint, then shift select the wristGrp. Choose `Constraints > Orient` with default options. This will cause the group node's local rotation axes to line-up perfectly with the wrist joint's axis. Now delete the constraint node from the hypergraph FIGURE 1.76. We only needed to constrain the wristGrp to line up the axes. The constraint is no longer needed.

Figure 1.76.



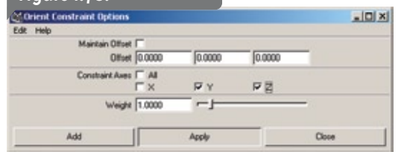
9. Now parent the armControl curve under the wristGrp group node. This will put garbage values in the armControl's transformations. This is OK. Now that the curve is parented under the wristGrp, we can freeze the transformations on it. After freezing the transformations, the curve's transform values will read zero and most importantly, it's local rotation axes will line-up with the wristGrp's axes which are lined up with the wrist joint's axes. You can verify this by looking at the rotation manipulator in the viewport FIGURE 1.77. It displays a different color for each axis. If these colors are aligned exactly the same on the joint and the curve, then the local rotation axes are lined up. This trick is very handy for connecting iconic representation throughout the body.

Figure 1.77.



10. Open exercise1.4_MiddleC.mb to see the control curve after it is properly aligned. With the axes properly lined up, we are ready to connect it to the forearm and wrist joints. Select the control curve and shift select the wrist joint, choose `Constrain > Orient`. Go into the options box and check the box beside Y and Z. Hit 'Apply' to create the constraint FIGURE 1.78. Recall that the X axis is the twisting axis. We want the twisting motion to be derived from the forearm, not the wrist. This is why we left the X axis out of the wrist constraint.

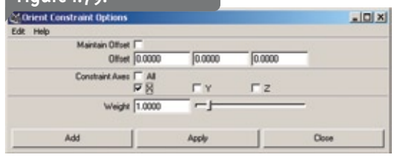
Figure 1.78.



11. To connect the controller to the forearm joint, we will create a constraint on only the X axis. Select the control curve, shift select the forearm joint and choose `Constrain > Orient`. Go into the options box and check the X axis. Hit 'Apply' to create the constraint FIGURE 1.79. This will cause the X axis of the forearm joint to rotate with the wrist controller.

12. Open exercise1.4_MiddleD.mb to see the wrist controller after being connected to the joints. Try rotating the wrist in each axis. You will notice that any twisting motion comes from the forearm while the wrist is still free to bend sideways and up/down. This effectively creates a separation of axes and allows the animator to animate the forearm and the wrist in one simple control.

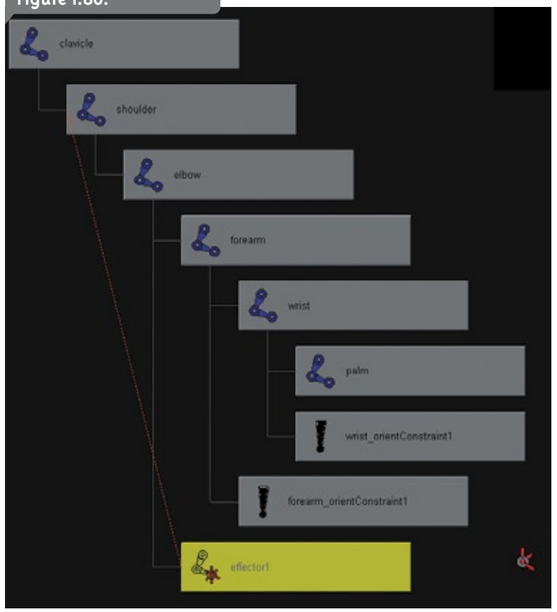
Figure 1.79.



13. Lastly, we have the problem of how to connect the IK handle to the wrist controller. Recall that the IK handle is at the forearm joint. We could have placed the IK handle at the wrist, but this would have caused the forearm to bend. So the problem now is how do we move the IK handle to the wrist so that it can be parented under the wrist control? Fortunately, there is a nice little trick that can do just that.

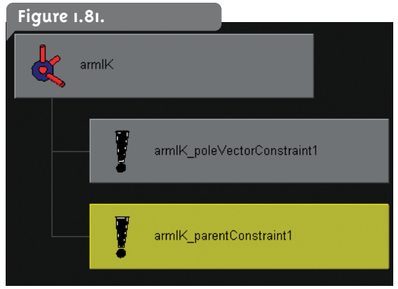
14. To move the IK handle, we first need to disable it so that the solver does not affect the joints while we move it. Select `Modify > Evaluate Nodes > Ignore All`. This will disable the IK handle. Now drop into the hypergraph and find the IK handle's end effector node. This node is created with the IK handle and should be parented underneath the elbow joint FIGURE 1.80. Select this node and activate the move tool. Hit the `insert` key to enter into pivot mode and snap the pivot of the end effector to the wrist joint. Do the same for the IK handle. When you are done, the IK handle and the end effector should be located at the wrist joint.

Figure 1.80.



15. With the IK handle moved into the proper position, we can turn the IK solver back on. Select `Modify > Evaluate Nodes > Evaluate All`. To connect the IK handle under the wrist controller, you can parent or parent constrain the IK handle to the wrist control curve FIGURE 1.81.

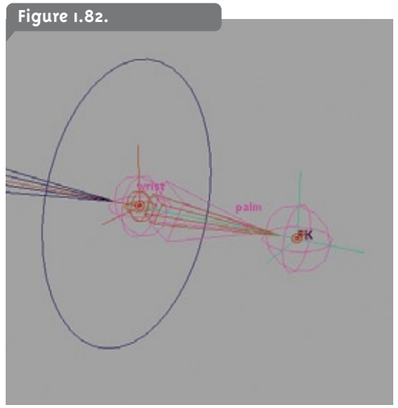
Figure 1.81.



16. Open exercise1.4_MiddleE.mb to see the setup with the connected IK handle. You can try translating and rotating the controller curve to pose the arm. Notice that the wrist, forearm and IK handle are all controlled by one single controller.

17. The arm rig is almost finished. You may notice that if you leave the setup as is, it will work fine except that the hand may slide around slightly as the rest of the body moves. Recall that the whole point of using IK is to be able to plant the hand on something. To prevent this sliding problem, create a single chain IK handle from the wrist to the palm joint FIGURE 1.82. Name this IK handle, 'palmIK'. This IK handle will stabilize the hand and allow the animator to firmly plant it on an object.

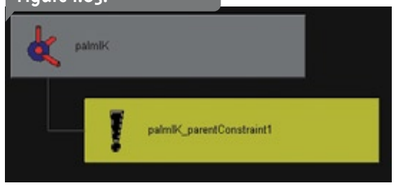
Figure 1.82.



18. Parent constrain the palmIK handle to the controller curve FIGURE 1.83.

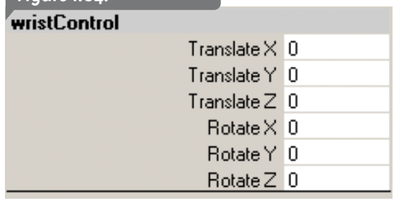
19. Open exercise1.4_MiddleF.mb to see the newly created, properly connected, IK handle. Grab the clavicle joint and rotate it around. You will notice that the hand remains firmly planted. When you add your finger joints into the hierarchy, be sure to parent them to this palm joint, not the wrist.

Figure 1.83.



20. To complete the setup, clean up the channels on the wrist controller. Select the scale and visibility channels and right-click to bring up the channel box menu. Choose 'Lock and Hide Selected' to remove these channels from the channel box FIGURE 1.84.

Figure 1.84.



21. Open exercise1.4_Finished.mb to play with the final rig.

With the automatic forearm setup, you can simulate the twisting motion of the forearm. When you bind the mesh to the joints, be sure to weight the forearm joint such that the skin twists around believably. For creatures with very long, or very fat forearms, you may find it better to use several (between 2-4) forearm joints. You can set this up in exactly the same way, except instead of controlling the twisting with an orient constraint, you can use an expression. This expression should be designed to divide the total twisting value across all of the forearm joints. To get a nice gradated falloff, setup the forearm joints to inherit less twisting, the closer they get to the elbow.

Also, be aware that this setup works quite well on an FK wrist as well. In fact, you can combine the automatic forearm with an IK/FK switch and even a no-flip elbow (just like the knee). All three of these setups can be combined to create a very versatile and robust arm setup.

Introduction to MEL Scripting:

What is MEL?

MEL stands for Maya's Embedded Language. MEL is an interpreted scripting language. This means that unlike lower-level languages like C++ or Java, MEL does not need to be compiled before you can run your program. A program made in MEL can be written and executed entirely within Maya. The language itself is loosely based on C syntax rules. Programmers who are familiar with C or C++ will be immediately familiar with MEL.

A MEL program is called a script. A MEL script is simply an ASCII text file with the .mel extension. To run a MEL script, open it with Maya's script editor and choose `Script > Execute`. Maya will parse through the text file and interpret the commands in the order with which they occur (from top to bottom). Almost everything you do in Maya is actually executing a MEL command in the background. Writing a MEL script is as simple as placing these commands into a text file so that you can then execute them as many times as you wish.

This type of script is called a macro. Macro's are the simplest types of scripts to make. To write a macro, you just need to find the commands that you want Maya to execute, and place them in into a script file. Then you can execute this script file as many times as you like.

Why Learn MEL?

So far, this chapter has focused on creating clean, robust creature rigs. While you have learnt some great techniques to setup creatures, you may have noticed that it can take some time to go through the actions. MEL provides a nice way to automate these sorts of setups. This allows you to complete these setups with the click of a button.

This may seem like a difficult task, and it can be. To create a macro script that will complete a specific setup, you must very carefully construct a script that mimics the exact commands that Maya is executing as you go through the setup tasks. The advantage being that when the script is finished, you can complete the setup as many times as you like, very quickly.

While this may seem like more work that it is worth, an auto-rigging script can save countless hours in a production that needs many characters to be rigged. In this section, we will dissect a macro script that will automate the process of creating a reverse foot with an IK/FK switch.

How Does a MEL Script Work?

Like was mentioned earlier, a MEL script is a text file. This script file is stored on the hard drive with a .mel extension FIGURE 1.85. A script file can contain any number of commands. When you execute a script, Maya reads through the file and executes each command, one after another.

Figure 1.85.

So what does a command look like? There are hundreds of commands in Maya, all of which are very well explained in the MEL command reference (`Help > MEL Command Reference`) FIGURE 1.86. There are commands to create objects, rename them, delete them, move them etc... There is a command in Maya for almost everything that you can do from the user interface, plus some that you cannot reach from the interface.

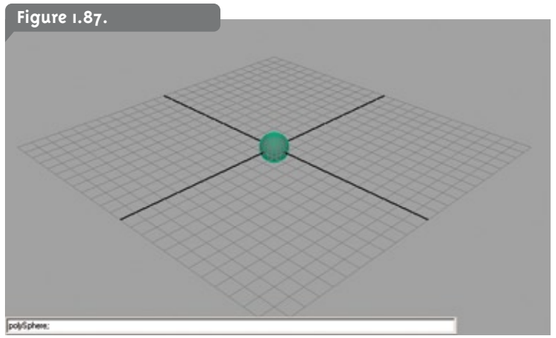
Figure 1.86.

The best way to get a handle on MEL commands is to start using them! Open Maya and type the following in the command line:

```
polySphere;
```

Hitting enter will cause Maya to read the command, interpret it and then execute it. The 'polySphere' command creates a polygon sphere. You should now have a polygon sphere in your scene FIGURE 1.87.

Figure 1.87.

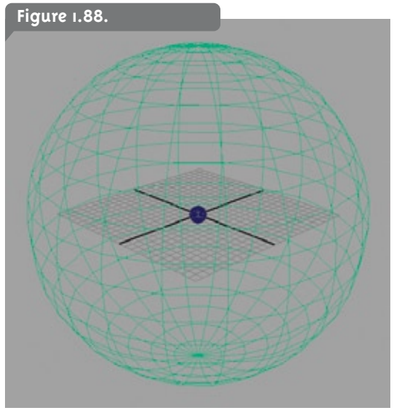


Commands also have what are called flags. Each command will have different flags and these flags are used to specify how you want the command to work. In the case of the polySphere command, there are eight flags that can be used. There is an explanation for each command and each of it's flags in the MEL command reference. Let's take a look at the '-radius' flag on the polySphere command. This flag controls the radius of the polygon sphere. To use this flag, type the following into the command line:

```
polySphere -radius 20;
```

If you hit enter, this command will create a much larger polygon sphere FIGURE 1.88. Notice that the command is followed by a white space, then the flag, then the value 20. The number 20 is known as the argument. Also notice that the entire command is on one line which is ended by a semi-colon. You must add a semi-colon to specify the end of a command. The semi-colon is called an end terminator.

Figure 1.88.

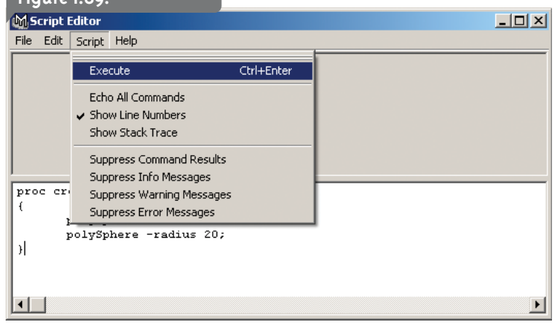


You can organize a group of commands within a script file to create what is called a procedure. This group of commands can then be called with it's own command. A procedure looks like this:

```
proc createTwoBigSpheres ()
{
        polySphere -radius 20;
        polySphere -radius 20;
}
```

This procedure is named 'createTwoBigSpheres'. To declare a procedure, you must use the reserved word 'proc' followed by the name of the procedure. The brackets that come after the name of the procedure are used to specify an argument for the procedure. Many procedures do not require arguments, in which case you include the brackets, but do not put anything in them. The createTwoBigSpheres procedure has no arguments. Later in this chapter, we will look at a procedure that does use an argument.

Figure 1.89.



The actual guts of the procedure are enclosed by the open/close curly braces. This procedure has two commands in it. Both of these commands will create a large sphere. Type this procedure into the script editor exactly as shown. If you now source this script (`Script > Execute`), you will notice that nothing happens FIGURE 1.89. A procedure will not do anything until it is called. To call a procedure, you can simply type the name of the procedure into command line and press enter. Type the following into the command line and press enter:

```
createTwoBigSpheres;
```

When you press enter, Maya will call the procedure named 'createTwoBigSpheres' and begin executing the commands inside the procedure. In this case, it will create two large spheres into the Maya scene.

Procedures are extremely useful. You can group commands together and call them from inside a script as many times as you wish. This allows you to create structured programs with many procedures that are reusable and flexible.

The MEL script examples in this book will concentrate, for the most part, on explaining the contents of specially made procedures and how these procedures interact with other procedures to create a useful program flow.

■ Data Types

The macro script we will be taking a look at shortly utilizes both commands and procedures. While you can do a lot with just commands and procedures, you will often find that it would be useful to have a way to store your own data. This is done using what is known as a variable.

There are three main variable types in MEL, they are:

Integers: These are whole numbers, meaning they cannot contain a decimal point. Integers are especially useful for counting things.

Floats: A float number is a floating point number. These are decimal numbers.

Strings: A string is a series of alphanumeric characters. A string variable can contain the names of nodes. There are many MEL commands that are specifically designed to modify strings.

To make a variable, you must declare it. Declaring a variable is a simple as specifying the type, followed by the name of the variable.

```
int $numberOfSpheres;
float $radiusOfSphere;
string $nameOfSphere;
```

With these variables declared, we can now assign values for them to hold. Think of a variable as a small chunk of memory that can hold data for you.

```
$numberOfSpheres = 3;
$radiusOfSphere = 2.1234567;
$nameOfSphere = "ball";
```

Notice that the string variable is assigned to the value between the quotation marks. We can now print these variables to the feedback line like this:

```
print $numberOfSpheres;
print $radiusOfSphere;
print $nameOfSphere;
```

In fact, you can use a variable in place of any argument. So to create a sphere with a radius of 2.1234567, you could execute this command:

```
polySphere -radius $radiusOfSphere;
```

The scripts throughout this book make heavy use of all three types of variables. You will also see how to do math operations on integer and float variables. Creating useful scripts often requires that you can manipulate string variables. This is also covered in many of the script examples. There are other data types besides integers, floats and strings. The vector and matrix data types are more specialized and will be discussed as they are encountered.

■ Arrays

Arrays are a special type of variable. One single array can be used to contain many different variables. These elements of the array are stored and retrieved with an index number. When you create an array, you can specify the maximum number of indices to include. Declaring an array is done like this:

```
int $numberOfSpheres[5];
float $radiusOfSphere[10];
string $nameOfSphere[];
```

Notice, the only difference between an array and a regular variable is the inclusion of the square brackets. These square brackets tell Maya that the variable is an array and that it contains X+1 number of indices where X is the number between the brackets. The array, $numberOfSpheres[5], actually contains 6 different indices. This is because arrays are zero-based. This means that the first element in an array is given the number 0. The second element is number 1, third is 2 and so on. If an array is declared with no number between the brackets (like $nameOfSphere[]), it is called a dynamic array. This array will grow in size to contain however many elements you need.

Assigning a value to a specific element in an array is done like this:

```
$radiusOfSphere[0] = 15;
```

This will make the first element equal to the value 15. You can assign the second element like this:

```
$radiusOfSphere[1] = 23;
```

You can print an entire array like this:

```
print $radiusOfSphere[];
```

Or only print a specific element:

```
print $radiusOfSphere[1] ;
```

Arrays are extremely handy for holding large collections of data. There are many examples throughout this book that will show you exactly how to use an array.

■ Further MEL

While this book contains many excellent examples of how to script useful functionality into Maya, it is not intended to be a complete guide to MEL scripting. For this reason, we assume at least a basic understanding of Maya's language. If you are coming from a programming background (like C, C++ or Java), you may find that this section is enough to get you started. If you are completely new to programming in general, you may find some of the concepts to be somewhat foreign. Do not panic, this stuff can be difficult to learn. There are many great and helpful texts that will take you from being a newbie to novice in no time. I would highly recommend that you pick up a MEL specific book if you have no previous programming experience. The next section will take the user through the process of creating a useful macro procedure. This is the easiest type of script to make, but they can be extremely useful and are a great way to learn MEL.

The cgTkSetupLeg MEL Script:

This is the first of many examples in this book that are designed to show you how a technical director can use MEL to create their own tools. The MEL script we are going to create in this section will enable you to setup an IK/FK switch and a reverse foot with the click of a button.

This entire script is stored into one single procedure, when this procedure is called, Maya will execute every command inside it. These commands will work on the current scene to automatically setup an IK/FK switch and a reverse foot. If you have not already read the exercises in this chapter that explains these setups, please do so before continuing. It is absolutely essential that you have a perfect understanding of the process you are trying to replicate. If you do not know the exact steps required to complete the setup, how can you expect to tell Maya what to do? Before we dive into the code, let's take a look at how the script works.

This script assumes that there are some objects that exist in your scene. These are the script's prerequisites. To use the cgTkSetupLeg procedure, you must create a joint chain for the leg. This chain must be named appropriately for the script to work. The cgTkSetupLeg_Start.mb scene file contains the exact prerequisites that are needed. Before executing the script, ensure that the following nodes (named exactly the same) are in the scene:

-lHip (joint)
-lKnee (joint)
-lAnkle (joint)
-lBall (joint)
-lToe (joint)
-leftFootControl (control object ie. NURBS curve or polygon)

The script will look for these nodes and do the setup on only these objects. If you want to use a macro script like this one, then you must be very careful about creating and naming the required objects exactly as the script needs them. To setup a right leg, create the same objects but rename them to replace the l's with r's and 'left' with 'right'.

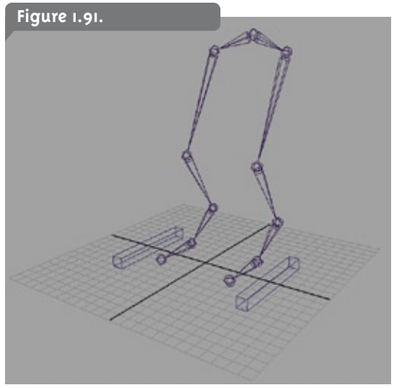
Let's take a quick look at exactly how to use this script:

1. Copy the cgTkSetupLeg.mel script file from the CD to your maya/scripts directory FIGURE 1.90.

Figure 1.90.

2. Restart Maya and open the cgTkSetupLeg_Start.mb scene file. This file contains all of the prerequisites for setting up a leg using this script. Notice that the joints have all been placed and named according to the conventions mentioned above FIGURE 1.91. There are also two polygon cubes in the scene. These will be used as the foot controllers.

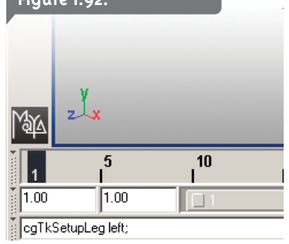
Figure 1.91.



3. Type 'cgTkSetupLeg left;' into the command line FIGURE 1.92. Hit enter to setup the left leg.

4. Type 'cgTkSetupLeg right;' into the command line. Hit enter to setup the right leg.

Figure 1.92.



5. Grab the foot controllers and notice that they now have the reverse foot custom attributes on them. The legs have also been setup with an IK/FK switch. When you executed the script's command, it found the properly named joints and automatically went through the entire setup procedures. Opening the hypergraph will reveal that the script has created many new nodes and arranged them in an intelligent manner FIGURE 1.93 .

Figure 1.93.

This kind of script is known as an auto-rigger. Most studios will have an auto rigger of their own that is tailor made to create rigs that work well for the animators and their productions. There are some excellent auto-rigging scripts available on the Internet, most of which have been created by individuals who were kind enough to share their work.

You may wonder then, if there are all these programs out there that automatically do the job of a technical director, then why do TD's still have jobs? The answer is simple, somebody has to create and maintain these programs. One auto-rigging script will never be able to handle every situation that is thrown your way. They are not designed to cover every single scenario that might arise, rather they are built for special cases. In the case of cgTkSetupLeg, this script was designed to quickly create a reverse foot and IK/FK switch for a normal bipedal creature.

It is not difficult to imagine a situation where a script like this would be extremely handy. A production may have anywhere from 10 to 100 different biped creatures that all need IK/FK switches and reverse foot setups. Rather than drive oneself insane by setting these up by hand, a TD can create a script that will automate the entire process. As a rule, if you have to do something more than two or three times, it is probably best to create a script.

The Code

This script is 318 lines long. The first 70 lines in the script file are dedicated to a header and error checking. Error checking is an important part of every script, but for the sake of clarity, we will not be covering the error checking in this script. The version of the script on the DVD includes several checks to ensure that all of the prerequisite objects exist in the current scene file. If they do not, an error is returned and the execution of the script is halted. This kind of error handling should always be done after you have got the core of the script working correctly. Error checking is covered in detail in later chapters.

Let's dig into our first script by dissecting each line in the order they appear in the file. While some more advanced scripts have a back/forth program flow, simple macro scripts like this one execute from top to bottom. This makes them very easy to read and a perfect introduction to MEL scripting in general.

Like was mentioned earlier, this script is contained within a single global procedure. This procedure is named the same as the script file (cgTkSetupLeg). To actually use this script, the user types the name of the main procedure into the command line followed by an argument to specify which side they wish to setup (left or right). When they execute this command, they are making a call to a global procedure. The first thing you need to do when starting your script is declare the main procedure you want to use. Let's do this now:

```
global proc cgTkSetupLeg(string $legSideLong)
{
        //Commands go here.
}
```

Notice that we have declared a procedure using 'proc'. The name of the procedure is 'cgTkSetupLeg'. This procedure has one argument. This argument is declared inside the parentheses. To declare a procedures argument, you must enter the data type (in this case 'string') followed by the name of the variable you want the argument stored into (in this case '$legSideLong').

You may be wondering about the word 'global' that appears before the procedure declaration. This makes the procedure a global procedure. Global procedures can be called from anywhere inside Maya. If you had neglected to make this procedure global (by just using 'proc'), you would have created a local procedure. Local procedures can only be called from the command line if you source the script.

This entire script is designed to work on either a left leg, or a right leg. In order to do this, the procedure uses two variables that contain left/right prefixes depending on which side the user has specified as an argument. These two string variables are $legSide and $legSideLong. The $legSide variable contains either 'l' or 'r'. The $legSideLong string contains the full name of the current side, either 'left' or 'right'. These variables are used throughout the entire script as prefixes to specify which object to work on. Notice in the procedure declaration, that the argument is stored into $legSideLong. This means that the $legSideLong variable contains the argument that the user tacked on to the end of the procedure call.

If the user calls the script with this:

```
cgTkSetupLeg left;
```

Then the $legSideLong variable will be equal to 'left'. In this case, we must make the $legSide variable contain a lowercase 'L'. If the user called the script and specified 'right', the $legSide variable should contain a lowercase 'R'. To decide which prefix to use, we will use some 'if' statements. These are also known as conditionals. Edit the procedure to look like this:

```
global proc cgTkSetupLeg(string $legSideLong)
{
        string $legSide;

        if ($legSideLong == "left")
        $legSide = "l";

        if ($legSideLong == "right")
        $legSide = "r";
}
```

We start off by declaring a new string variable called $legSide. Then we check to see what argument the user specified (left or right). The 'if' statement checks whether the expression inside the brackets is true or false. If it evaluates to true, then the line immediately below the if statement is executed. If it evaluates to false, then the interpreter skips the next line and continues with the rest of the script.

With this example, you have been shown how to use arguments with procedures. While arguments can be extremely useful, please understand that not every procedure requires an argument. There are many examples in this book of procedures that do not require arguments. Use an argument when you want to pass some information into a procedure. In this case, we needed to tell the procedure which leg to setup, the left or the right.

Before we get into the actual commands to start setting things up, we have some other variables that we must get out of the way. Recall from the reverse foot setup that we must create some set driven keys to control the behavior of the group nodes. Doing so requires that we know how the joints are oriented. This script assumes that the user is using default orientations with the leg chains pointing in the positive Z direction. This is how the leg chains should be created. Most productions use a standard where the creature's default pose is standing on the origin and facing the positive Z direction. So let's create some variables to declare these conventions.

```
$footAxis = "z";
$isFootNegative = -1;       //Change this to '1' if foot axis is
                            //positive when rotated down.
```

These variables are used near the end of the reverse foot setup. For now, just remember that this is where they are and that they are positioned at the top of the script if you need to change them for some reason. Also notice that I did not declare these variables. MEL is a very user friendly language and it will try to determine what type of variables to create if you do not declare the type explicitly. This is called automatic type declaration. While it is usually fine to do this, realize that this can cause some difficult to catch bugs at times. It is never a bad habit to define every variable in your script.

The IK/FK Switch

Phew! We're finally ready to start the setup. From here on, the procedure is going to start creating an IK/FK switch. Once this is done, the second half of the procedure will setup the reverse foot. The following code can be placed directly below the last snippet. To start with, we must orient the joints.

```
//ORIENT JOINTS
select -r ($legSide + "Hip");
joint -e -oj xyz -ch -zso;
```

To orient a chain of joints in Maya, it is easiest to select the root and execute `Skeleton > Orient Joint`. This menu command actually executes the 'joint' command. This little snippet of code mimics this action exactly. The first command is 'select'. This command will select any object you like. The '-r' flag specifies that the selection should replace anything that is currently selected. In this case we want to select the hip joint. This hip joint is either named 'lHip' or 'rHip'. To select the right hip joint, we concatenate the $legSide variable with the string "Hip". If the user specified the 'left' argument, the $legSide variable will now be equal to 'l'. This would make the first line above select the object in the scene named 'lHip'. This is the root joint in our leg hierarchy.

Now that we have the leg selected, we can orient the joints using the 'joint -e -oj xyz -ch -zso;' command. This line may look like a pile of alphabetic puke, but it is actually just a single command, with four flags. The first flag, '-e' , means '-edit'. This puts the command in edit mode. Without this flag, the command is used to create joints. We do not want to create any joints, we just want to orient some existing ones, so we use the '-e' flag. The next flag, '-oj' stands for '-orientJoint'. This flag requires a string argument that specifies the order that we want the axes to be oriented in. In this case, we chose 'xyz' order. This is the default Maya joint orientation and makes the X axis the twisting axis. The third flag is the '-ch' or '-children' flag. This tells the command to apply any edits to all of the children joints. The fourth and final flag is '-zso'. This flag sets the scale orientation to zero.

If this alarming number of commands and flags appears intimidating, don't panic! There is no need to memorize all of this stuff. When you write your own scripts, use the MEL command reference to determine what flags, if any, you wish to attach to your commands. If you cannot determine which flags to attach, try executing the command from the Maya interface and watch in the script editor to see what flags are used. The script editor and the MEL command reference are all that is needed to write a macro script like this one.

After orienting the joints, we must duplicate them twice to create the FK and IK joint chains.

```
//DUPLICATE JOINT HIERARCHY
select -r ($legSide + "Hip");
duplicate -rr;
duplicate -rr;
```

In this case, we select the hip joint again, and execute the duplicate command. This is the same command that is executed from the `Edit > Duplicate` menu. With two duplicate commands, we will duplicate the joint chain twice. This creates an IK chain and an FK chain. The '-rr' flag is the shortened version of '-returnRootsOnly'. This is just the default flag that is attached to the duplicate procedure when you do not want the duplicates to have any connections to the original object.

With the duplicates made, we need to rename the FK and IK chains. To start with, we can rename the FK chain so that they are all named like this:

lHipFK
lKneeFK
lAnkleFK
lBallFK
lToeFK

To rename these joints, we use the rename command. The rename command takes two arguments. The first argument is the current name of the object, the second argument is the name that you want to change it to. To rename a polygon sphere named pSphere1, it would look like this:

```
rename pSphere1 ball;
```

This will name the sphere 'ball'. To rename our joints, however, we must be a little bit more specific. Because Maya will allow two objects in different coordinate spaces to be named the same thing, it uses something called an absolute name to refer to an object that is a child of other objects. When we duplicated the joint hierarchy, we created three versions of each joint (except the root), all with the same name. So there are three nodes in the Maya scene named lAnkle. To rename each of the newly created joints, we need a way of referring to them. So in order to specify which joint we want to rename, we must use what is known as an absolute name. This is the name of an object as it relates to it's hierarchy. For instance, if we parent a sphere under a group named 'group', then the actual name of the sphere is 'group|pSphere1'. If we parent a cone underneath the sphere, the cone's absolute name is then 'group|pSphere1|cone'. We must refer to objects that are in a hierarchy by their absolute names. This prevents naming conflicts.

To rename the FK and IK joint hierarchies, we use the absolute names to avoid naming conflicts.

```
rename ($legSide + "Hip1") ($legSide + "HipFK");
rename ($legSide + "HipFK|" + $legSide + "Knee") ($legSide + ...
        "KneeFK");
rename ($legSide + "HipFK|" + $legSide + "KneeFK|" + $legSide + ...
        "Ankle") ($legSide + "AnkleFK");
rename ($legSide + "HipFK|" + $legSide + "KneeFK|" + $legSide +...
        "AnkleFK|" + $legSide + "Ball") ($legSide + "BallFK");
rename ($legSide + "HipFK|" + $legSide + "KneeFK|" + $legSide + ...
        "AnkleFK|" + $legSide + "BallFK|" + $legSide + "Toe") ...
        ($legSide + "ToeFK");
```

These six rename commands will rename one of the duplicate joint chains to tack on 'FK' to the end of the name. Renaming the IK chain is done in the same way:

```
rename ($legSide + "Hip2") ($legSide + "HipIK");
rename ($legSide + "HipIK|" + $legSide + "Knee") ($legSide + ...
        "KneeIK");
rename ($legSide + "HipIK|" + $legSide + "KneeIK|" + $legSide + ...
        "Ankle") ($legSide + "AnkleIK");
rename ($legSide + "HipIK|" + $legSide + "KneeIK|" + $legSide + ...
        "AnkleIK|" + $legSide + "Ball") ($legSide + "BallIK");
rename ($legSide + "HipIK|" + $legSide + "KneeIK|" + $legSide + ...
        "AnkleIK|" + $legSide + "BallIK|" + $legSide +"Toe") ...
        ($legSide + "ToeIK");
```

With the duplicated and properly named joint hierarchies, we can begin to build the IK/FK switch. Recall from the setup that we must orient constrain each joint in the original chain to both the IK and FK equivalents. In Maya, an orient constraint is accessed from the Constraint menu. In the background, this menu executes the 'orientConstraint' MEL command. To setup some constraints with MEL, we will replicate the exact same process that you would use in the interface. That is, select the driver object, shift select the target and create the constraint. To create a constraint between the lHipFK and the lHip joints, we must select the lHipFK, then shift select the lHip and execute the command. In MEL, it looks like this:

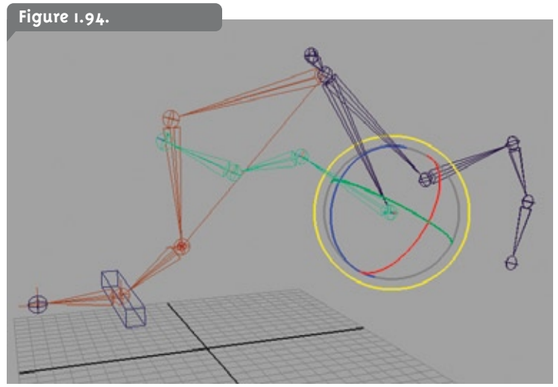
```
//CREATE ORIENT CONTRAINT
select -r ($legSide + "HipFK") ;
select -tgl ($legSide + "Hip") ;
orientConstraint -offset 0 0 0 -weight 1;
```

Notice the use of the '-tgl' flag on the second select command. This flag mean '-toggle' and is the same as holding down the shift key. It will add the object to the current selection list. Again, we are using the $legSide variable to distinguish between the left/right leg. The orientConstraint command has only two flags. The first flag, '-offset' specifies the offset in each axis in degrees. Because these joints are perfectly aligned duplicates, we want no offset at all (hence, 0 0 0). The '-weight' command specifies the strength of the constraint. A value of 1 is 100%.

We must constrain each joint (except the toe) to it's corresponding IK and FK versions. The rest of these constraints are setup like this:

```
select -r ($legSide + "HipIK") ;
select -tgl ($legSide + "Hip") ;
orientConstraint -offset 0 0 0 -weight 1;

select -r ($legSide + "KneeFK") ;
select -tgl ($legSide + "Knee") ;
orientConstraint -offset 0 0 0 -weight 1;

select -r ($legSide + "KneeIK") ;
select -tgl ($legSide + "Knee") ;
orientConstraint -offset 0 0 0 -weight 1;

select -r ($legSide + "AnkleFK") ;
select -tgl ($legSide + "Ankle") ;
orientConstraint -offset 0 0 0 -weight 1;

select -r ($legSide + "AnkleIK") ;
select -tgl ($legSide + "Ankle") ;
orientConstraint -offset 0 0 0 -weight 1;

select -r ($legSide + "BallFK") ;
select -tgl ($legSide + "Ball") ;
orientConstraint -offset 0 0 0 -weight 1;

select -r ($legSide + "BallIK") ;
select -tgl ($legSide + "Ball") ;
orientConstraint -offset 0 0 0 -weight 1;
```

Figure 1.94.



A common problem with orient constraints is that of flipping. Sometimes the legs and arms will flip instead of blending smoothly between IK and FK FIGURE 1.94. This can usually be fixed by changing the interpolation type on the orient constraint. This can be done from the channel box with the orient constraint selected, or you can do it in MEL like this:

```
//CHANGE INTERPOLATION TYPES ON ORIENT CONSTRAINTS
setAttr ($legSide + "Hip_orientConstraint1.interpType") 2;
setAttr ($legSide + "Knee_orientConstraint1.interpType") 2;
setAttr ($legSide + "Ankle_orientConstraint1.interpType") 2;
setAttr ($legSide + "Ball_orientConstraint1.interpType") 2;
```

This introduces us to one of the most useful MEL commands, the 'setAttr' command. This is used for setting the value of any attribute on any object in your scene. The command takes two arguments, the first is the name of the object.attribute, followed by the value that you want to set it to. For example, to move a ball up two units in Y, would look like:

```
setAttr ball.translateY 2;
```

In the case of the interpolation types, they have all been set to '2'. This is the interpolation method known as 'Shortest'. This forces the joints to blend in the shortest distance possible. This usually prevents the flipping problem.

With the joint chains setup and orient constrained, the only thing left to do is setup the set driven keys to connect the orient constraints to the controller object. These set driven keys will be controlled by driver attributes on the footControl object. Recall that this object was one of the prerequisites. In order to begin the set driven key setup, this controller object must have some custom attributes added to it. Recall from the setup exercise that we did this with the Add Attribute window. Doing it in MEL is actually pretty straightforward. Adding an attribute is done with the addAttr MEL command. The following code will add six attributes that are needed for both the reverse foot and the IK/FK switch. Even though we haven't started the reverse foot yet, it is more convenient to add all of the necessary attributes in one place.

```
//ADD ATTRIBUTES FOR IK / FK SWITCH AND REVERSE FOOT
addAttr -ln toeTap -k 1 -at double  -min -10 -max 10 -dv 0 ...
        ("|" + $legSideLong + "FootControl");
addAttr -ln peelHeel -k 1 -at double  -min 0 -max 10 -dv 0 ...
        ("|" + $legSideLong + "FootControl");
addAttr -ln standTip -k 1 -at double  -min 0 -max 10 -dv 0 ...
        ("|" + $legSideLong + "FootControl");
addAttr -ln twistHeel -k 1 -at double -min -10 -max 10 -dv 0 ...
        ("|" + $legSideLong + "FootControl");
addAttr -ln twistToe -k 1 -at double  -min -10 -max 10 -dv 0 ...
        ("|" + $legSideLong + "FootControl");
addAttr -ln ikFk -k 1 -at double  -min 0 -max 10 -dv 0 ...
        ("|" + $legSideLong + "FootControl");
```

The addAttr command uses five different flags, but it is easy to distinguish their meanings just by looking at it. The -ln flag specifies the name of the attribute. The -k flag is used to make the attribute keyable (a value of 1 is keyable, 0 is non-keyable). The -at flag specifies the type of attribute. In this case, we are using a 'double' which is a decimal value, like a float. The -min flag specifies the minimum value the attribute can have, and the -max specifies the maximum. Finally, the -dv flag specifies the default value. This is the value the attribute will have when it is first created. When I constructed these commands, I simply added an attribute to an object using the regular Maya window while watching the output in the command feedback. This showed me exactly what flags I would need.

With the attributes added, we are ready to move onto the final part of the IK/FK switch setup. This is where we connect the custom attributes to the weight of the orient constraints. In this way, we can setup the constraints to blend between the FK leg and the IK leg.

Making set driven key connections in MEL is quite straightforward. For simplicity sake, we can script it exactly how it is done is Maya. By setting the driver object value first, then setting the driven object value, then making the key. To start with, we will set a driven key for the hip orient constraint. Recall from exercise1.2 that we must set two keys for each orient constraint on each joint. These keys connect the weight value of the constraint to the custom attribute that we just added to the controller curve. To start with, we will use the setAttr command to set the driver and the driven values:

```
setAttr ($legSideLong + "FootControl.ikFk") 0;
setAttr ($legSide + "Hip_orientConstraint1." + $legSide + "HipIKW1") 1;
setAttr ($legSide + "Hip_orientConstraint1." + $legSide + "HipFKW0") 0;
```

With the driver and driven values set, we can set the driven key. This is done with the setDrivenKeyframe MEL command:

```
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk")  ...
($legSide + "Hip_orientConstraint1." + $legSide + "HipFKW0");
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
($legSide + "Hip_orientConstraint1." + $legSide + "HipIKW1");
```

The setDrivenKeyframe command is followed by a single flag '-cd'. This flag takes an object's attribute as an argument. In this case, we feed it either 'leftFootControl.ikFk' or 'rightFootControl.ikFk', depending on the current value of the $legSideLong variable. Finally, we specify the name of the driven attribute. In this case, it is the name of the orient constraint that was made on the leg. After these two commands are executed, the set driven key connection will be made.

We have many more set driven key connections to make. This is a great

example of how MEL can be so much more efficient for tedious setup routines. By copying and pasting the above snippets of code, we can adjust the values to create all the driven keys we need. The following MEL commands will create all of the driven key connections that are specified in exercise1.2.

```
setAttr ($legSideLong + "FootControl.ikFk") 10;
setAttr ($legSide + "Hip_orientConstraint1." + ...
        $legSide + "HipIKW1") 0;
setAttr ($legSide + "Hip_orientConstraint1." + ...
        $legSide + "HipFKW0") 1;
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
      ($legSide + "Hip_orientConstraint1." + $legSide + "HipFKW0");
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
      ($legSide + "Hip_orientConstraint1." + $legSide + "HipIKW1");

setAttr ($legSideLong + "FootControl.ikFk") 0;
setAttr ($legSide + "Knee_orientConstraint1." + $legSide + ...
        "KneeFKW0") 0;
setAttr ($legSide + "Knee_orientConstraint1." + $legSide + ...
        "KneeIKW1") 1;
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
    ($legSide + "Knee_orientConstraint1." + $legSide + "KneeFKW0");
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
    ($legSide + "Knee_orientConstraint1." + $legSide + "KneeIKW1");

setAttr ($legSideLong + "FootControl.ikFk") 10;
setAttr ($legSide + "Knee_orientConstraint1." + $legSide + ...
        "KneeFKW0") 1;
setAttr ($legSide + "Knee_orientConstraint1." + $legSide + ...
        "KneeIKW1") 0;
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
    ($legSide + "Knee_orientConstraint1." + $legSide + "KneeFKW0");
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
    ($legSide + "Knee_orientConstraint1." + $legSide + "KneeIKW1");

setAttr ($legSideLong + "FootControl.ikFk") 0;
setAttr ($legSide + "Ankle_orientConstraint1." + ...
        $legSide + "AnkleFKW0") 0;
setAttr ($legSide + "Ankle_orientConstraint1." + ...
        $legSide + "AnkleIKW1") 1;
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
  ($legSide + "Ankle_orientConstraint1." + $legSide + "AnkleFKW0");
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
  ($legSide + "Ankle_orientConstraint1." + $legSide + "AnkleIKW1");

setAttr ($legSideLong + "FootControl.ikFk") 10;
setAttr ($legSide + "Ankle_orientConstraint1." + ...
        $legSide + "AnkleFKW0") 1;
setAttr ($legSide + "Ankle_orientConstraint1." + ...
        $legSide + "AnkleIKW1") 0;
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
  ($legSide + "Ankle_orientConstraint1." + $legSide + "AnkleFKW0");
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
  ($legSide + "Ankle_orientConstraint1." + $legSide + "AnkleIKW1");

setAttr ($legSideLong + "FootControl.ikFk") 0;
setAttr ($legSide + "Ball_orientConstraint1." + $legSide ...
        + "BallFKW0") 0;
setAttr ($legSide + "Ball_orientConstraint1." + $legSide ...
        + "BallIKW1") 1;
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
    ($legSide + "Ball_orientConstraint1." + $legSide + "BallFKW0");
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
    ($legSide + "Ball_orientConstraint1." + $legSide + "BallIKW1");

setAttr ($legSideLong + "FootControl.ikFk") 10;
```

```
setAttr ($legSide + "Ball_orientConstraint1." + $legSide + ...
        "BallFKW0") 1;
setAttr ($legSide + "Ball_orientConstraint1." + $legSide + ...
        "BallIKW1") 0;
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
    ($legSide + "Ball_orientConstraint1." + $legSide + "BallFKW0");
setDrivenKeyframe -cd ($legSideLong + "FootControl.ikFk") ...
    ($legSide + "Ball_orientConstraint1." + $legSide + "BallIKW1");
```

Finally, after all of the driven keys have been created, we can set the driver attribute back to zero. It is always a good idea to have your rig in it's default state after it is created. Doing this is as simple as setting the IK/FK switch attribute back to 'o'.

```
setAttr ($legSideLong + "FootControl.ikFk") 0;
```

This completes the IK/FK switch script. If you were to stop here, this script would already automatically create an IK/FK switch. On it's own, this is a very useful setup. To complete the script, let's have it also create a reverse foot like from exercise1.3.

■ Scripting the Reverse Foot

This next section will describe the rest of the cgTkSetupLeg procedure. This is not a separate script, but rather a continuation of the previous code. You can combine both of these setups into one single, very long procedure. The IK/FK switch took about 140 lines of code. The reverse foot is about 100 lines of code. This section requires many of the same principles that we introduced in the last section (set driven keys, selecting objects and the setAttr command). In addition to this, we will be exploring some more advanced commands.

If you haven't already, please read exercise1.3. It is essential that you understand the process we are trying to duplicate with MEL. Recall that a reverse foot setup is done on a leg chain with an ankle, ball and toe. This entire section will describe operations that are performed on the newly created IK joint chain. Recall that we had to duplicate the original joint chain to create the IK and FK copies. The reverse foot is created on the IK chain. To begin with, we must create a single chain IK handle from the hip to the ankle, from the ankle to the ball and from the ball to the toe.

To create some IK handles, we will replicate the exact procedure that you would use in the viewport. Start by selecting the start and end joints, then create the IK handle. This is done with the ikHandle MEL command.

```
select -r ($legSide + "AnkleIK.rotatePivot") ;
select -add ($legSide + "BallIK.rotatePivot") ;
ikHandle -n ($legSide + "BallIk") -sol ikSCsolver -s 0;
```

We start by selecting the ankle joint, then we use the '-add' flag to shift select the ball joint. The ikHandle MEL command has three flags. The first, '-n' flag, is used to specify a name for the IK handle. We must specify a name for the IK handle so that we know what to call it later in the script. The '-sol' flag specifies the type of solver you would like to create. In this case, we need a single chain IK solver. The options for these types of flags are covered in great detail in the MEL command reference. The last flag is '-s' which stands for '-sticky'. A sticky IK handle will cause the child joints to remain fixed as the parent is moved. In this case, we are controlling every joint in the foot with IK, so stickiness is not necessary on these IK handles, hence we specify a o to turn it off.

All boolean flags can be specified with either a 1 or a 0 for true/false or on/off.

The last snippet of code created an IK handle from the ankle to the ball. There are two other IK handles that we must create. They are from the ball to the toe and from the hip to the ankle:

```
select -r ($legSide + "BallIK.rotatePivot") ;
select -add ($legSide + "ToeIK.rotatePivot") ;
ikHandle -n ($legSide + "ToeIk") -sol ikSCsolver -s 0;

select -r ($legSide + "HipIK.rotatePivot") ;
select -add ($legSide + "AnkleIK.rotatePivot") ;
ikHandle -n ($legSide + "LegIk") -sol ikRPsolver -s 1;
```

These IK handles are created in exactly the same way. Notice that the IK handle from the hip to the ankle has had it's stickiness turned on (with the '-s' flag).

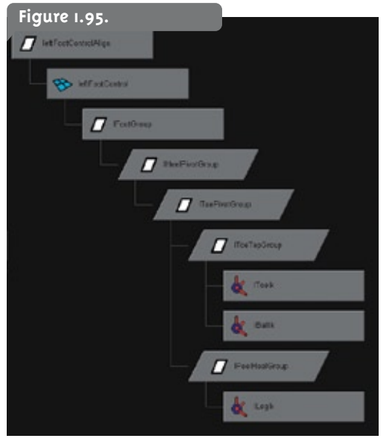
The next step in creating a reverse foot is to group these IK handles into a hierarchy that will allow us to control the foot from several different pivot points. While this step may be the most difficult to comprehend, grouping is very easy in MEL. To group an object in MEL, you can simply select the object(s) and execute the MEL command 'group'. To start with, we must group the toe IK and the ball IK:

```
select -r ($legSide + "ToeIk") ($legSide + "BallIk") ;
group -n ($legSide + "ToeTapGroup");
```

The select command is used to select both of the IK handles. The group command has one flag and that is '-n'. This flag needs a string to specify the name of the group. The rest of the grouping is handled in exactly the same way. By following the exact same procedure from exercise1.3, we can group the IK handles into a proper hierarchy:

```
select -r ($legSide + "LegIk") ;
group -n ($legSide + "PeelHeelGroup");

select -r ($legSide + "ToeTapGroup") ($legSide + "PeelHeelGroup") ;
group -n ($legSide + "ToePivotGroup");

group -n ($legSide + "HeelPivotGroup");

group -n ($legSide + "FootGroup");
```

Figure 1.95.



This creates five group nodes that are perfectly arranged to control the foot IK handles FIGURE 1.95. The only problem now, is that these group nodes do not yet pivot from the correct position in world space. When you create a new group in Maya, this group's pivot point will be located at the origin. To move these pivot points in exercise1.3, we simply snapped them to the ball and toe joints. This works fine, but unfortunately, there is no 'snap' command in MEL. This is one of the few cases where a macro script cannot directly imitate a setup procedure. To move these pivot points into the proper location, we must first find the world space coordinates for the ankle, ball and toe. To do this, we will use the 'xform' MEL command. The xform command is extremely useful. It can be used to gain transformation information about any objects in your Maya scene. In this case, we must query the world space vector of the foot joints. To do so, we are going to create a float array. This array will house the x, y and z components of the world space vector.

```
float $worldSpaceOfJoint[];
```

Having declared this float array, we can now put the necessary data into it. Let's start by finding the world space location of the ball joint.

```
$worldSpaceOfJoint= `xform -q -ws -rp ($legSide + "Ball")`;
```

Notice that we used the backticks to store the returned value of the xform command into the $worldSpaceOfJoint array. The xform command is a very important MEL command and can be used in many instances. In this case, we are putting the command into query mode with the '-q' flag. Commands in query mode will return data about the specified object. In this case, we want to find the world space coordinates of the rotation pivot. The '-ws' flag specifies world space and the '-rp' flag specifies the rotational pivot. The last argument (in brackets) is the name of the object that we want to query, in this case, the ball joint.

It may not be obvious, but the $worldSpaceOfJoint array now contains the x, y and z coordinates of the ball joint's rotation pivot. It does this by storing each coordinate into a separate element in the array. This means that the ball joint's X component will be stored into $worldSpaceOfJoint[0]. The Y component is stored in $worldSpaceOfJoint[1], and the Z is stored into $worldSpaceOfJoint[2].

Now we need to move the group node's pivot. To move the pivot point, we can use the 'move' MEL command. This command requires three floats (to specify the X, Y and Z components) followed by the name of one or more objects that you want to move. A typical move command may look like, 'move -a 4 5 3 pSphere1;'. This would move the object named 'pSphere1', to the world space location [4,5,3]. The '-a' flag specifies an absolute transformation. Recall that we want to move the pivot point of the toe tap group to the ball joint. This way, the toe joint will rotate from the ball.

```
move -absolute $worldSpaceOfJoint[0] $worldSpaceOfJoint[1] ...
      $worldSpaceOfJoint[2] ($legSide+ "ToeTapGroup.scalePivot")...
      ($legSide + "ToeTapGroup.rotatePivot") ;
```

Notice that we specified the x, y, and z components of the move by using the $worldSpaceOfJoint array. The objects we want to move are the toe tap group's scale and rotate pivots. This does not move the group itself, but just the point in space from which it rotates. The rest of the group node's pivots can be adjusted using the xform and move MEL commands:

```
$worldSpaceOfJoint= `xform -q -ws -rp ($legSide + "Ball")`;
move -a $worldSpaceOfJoint[0] $worldSpaceOfJoint[1] ...
  $worldSpaceOfJoint[2] ($legSide + "PeelHeelGroup.scalePivot") ...
  ($legSide + "PeelHeelGroup.rotatePivot") ;

$worldSpaceOfJoint= `xform -q -ws -rp ($legSide + "Toe")`;
move -a $worldSpaceOfJoint[0] $worldSpaceOfJoint[1] ...
  $worldSpaceOfJoint[2] ($legSide + "ToePivotGroup.scalePivot") ...
  ($legSide + "ToePivotGroup.rotatePivot") ;

$worldSpaceOfJoint= `xform -q -ws -rp ($legSide + "Ankle")`;
move -a $worldSpaceOfJoint[0] $worldSpaceOfJoint[1] ...
 $worldSpaceOfJoint[2] ($legSide + "HeelPivotGroup.scalePivot") ...
 ($legSide + "HeelPivotGroup.rotatePivot") ;

$worldSpaceOfJoint= `xform -q -ws -rp ($legSide + "Ball")`;
move -a $worldSpaceOfJoint[0] $worldSpaceOfJoint[1] ...
     $worldSpaceOfJoint[2] ($legSide + "FootGroup.scalePivot") ...
     ($legSide + "FootGroup.rotatePivot") ;
```

This completes the setup on the IK handles and their groups. From here, we must attach the foot controller object and hook-up the custom attributes to control the rotation of the IK group nodes. To attach the foot controller, we must follow a series of steps. This ensures that the foot controller object is properly aligned with the foot joints.

The first thing we do is group the foot control object to itself. This creates an extra group node that we can use to align with the joint while keeping the foot controller with zeroed transformations.

```
group -name ($legSideLong + "FootControlAlign") ($legSideLong ...
        + "FootControl");
```

Next, we must move the foot controller into position. Most animators prefer to control a foot from the ball joint. For this reason, we will snap the foot controller to the ball. We must also orient the controller so that it has the same orientation as the ball joint.

```
delete `orientConstraint ($legSide + "Ball") ($legSideLong ...
        + "FootControlAlign")`;
delete `pointConstraint ($legSide + "Ball") ($legSideLong ...
        + "FootControlAlign")`;
```

The lines above may look slightly odd. Notice the point and orient constraints are done inside back ticks. These back ticks encompass the entire argument for the delete command. This has the same effect as creating the constraint and then deleting it. The orient constraint is needed to align the local rotation axes, but then we can delete it. The point constraint is needed to move the foot controller to the location of the ball joint, but then it is no longer needed.

I have avoided using combined commands like this for the sake of making the code easier to read and understand. The important idea here is that when you become familiar with MEL, you may find it easier to combine commands by using back ticks to specify a return value as an argument for another command.

To make the foot controller actually control the foot IK, we must parent our reverse foot hierarchy under the foot controller. This is done with the 'parent' MEL command:

```
parent ($legSide + "FootGroup") ($legSideLong + "FootControl");
```

Now we can freeze the transformations on the control object to zero it out:

```
select -r ($legSideLong + "FootControl");
FreezeTransformations;
```

To finish the controller object, we must set the pole vector X, Y and Z components on the leg IK handle to 0. This will alleviate any odd IK behavior that may arise from creating the IK handles in a script:

```
setAttr ($legSide + "LegIk.poleVectorX") 0;
setAttr ($legSide + "LegIk.poleVectorY") 0;
setAttr ($legSide + "LegIk.poleVectorZ") 0;
```

Recall from exercise1.3 that the final step in creating a reverse foot is connecting the controller object's custom attributes to the rotation of the group nodes. This can be done many different way, but perhaps the easiest is using set driven keys.

The exact values for the reverse foot set driven keys are explained in exercise1.3. To setup them up with MEL, we will use the exact same technique that we used to connect the IK/FK switch to the controller attribute. That is, set the driver attribute, set the driven attribute, then create the key. There are either two or three keys that must be made for each attribute on the controller object. To start with, let's create the set driven keys for controlling the toe tapping motion:

```
setAttr ($legSideLong + "FootControl.toeTap") 0;
setAttr ($legSide + "ToeTapGroup.rotate" + $footAxis) 0;
setDrivenKeyframe -cd ($legSideLong + "FootControl.toeTap") ...
         ($legSide + "ToeTapGroup.rotate" + $footAxis);

setAttr ($legSideLong + "FootControl.toeTap") 10;
setAttr ($legSide + "ToeTapGroup.rotate" + $footAxis) ...
         (90*$isFootNegative);
setDrivenKeyframe -cd ($legSideLong + "FootControl.toeTap") ...
         ($legSide + "ToeTapGroup.rotate" + $footAxis);

setAttr ($legSideLong + "FootControl.toeTap") -10;
setAttr ($legSide + "ToeTapGroup.rotate" + $footAxis) ...
         (-90*$isFootNegative);
setDrivenKeyframe -cd ($legSideLong + "FootControl.toeTap") ...
         ($legSide + "ToeTapGroup.rotate" + $footAxis);

setAttr ($legSideLong + "FootControl.toeTap") 0;
```

These three driven keys will hook up the toe tap group's rotate Z to the custom toe tap attribute on the foot control. Notice that the last line resets the controller attribute back to zero. Be sure to write your auto-setup scripts such that the rig is created in a default pose.

There are ten more driven keys that must be set for the other four attributes on the controller object. These keys are described in exercise1.3. To script these, just copy/paste the above snippet of code and edit the driver/driven attribute names and values. This is a very quick way to script set driven key connections. They all use the same format, so just copy and paste to setup the code very quickly.

These set driven keys are the final step in the reverse foot setup. The only thing to do after this is print out some feedback to the user:

```
print ("The " + $legSideLong + " leg has been setup. \n");
```

■ The Finished Script

There are two scene files on the DVD that contain a start and finished copy of the leg setup we just finished scripting. This is a great script to learn MEL with. By learning from a macro, you have seen how everything in Maya can be interpreted as a series of commands. These commands can then be recorded and stored into a text file to create a script.

These kinds of scripts are very popular in every production. They can be used to give a production a significant speed boost and create perfect rig setups, with very little risk of human error.

Final Thoughts:

This chapter presented some interesting ideas for creating robust creature rigs in record time. By automating common processes, you can concentrate on the more important details. Spending countless hours on a tedious, repetitive task is on of the quickest ways to burnout in this industry. By using MEL to automate the boring stuff, you can spend your time designing amazing creatures to wow your audience.

Designing the perfect rig is tricky business. While this chapter has introduced you to the fundamentals of rigging, the rest of this book will hone those skills in key areas. Rigging is perhaps the most underrated discipline in 3d animation. This is largely due to the fact that most 3d artists never get past the ideas presented in this first chapter. They may experiment with joints and smooth skinning for a couple days before giving up and choosing to let someone else handle the 'boring' stuff. Chances are that if you are reading this, you will agree that rigging is not boring. A well made rig is like a carefully tuned machine. It can be a brilliant experience to watch as your machine matures into an elegant puppet with the ability to come to life. To continue the search for the perfect rig, continue reading in chapter 2 for an introduction to one of the most interesting principles in 3d animation, overlapping action.

Chapter 2
Squash and Stretch

Introduction to Squash and Stretch in Computer Animation:

The first chapter introduced one of the fundamental principles of animation, overlapping action. It might be hard to imagine, but the twelve original principles of animation were arrived at only after much hard work and experimentation by the pioneers at Disney and to a lesser extent, Warner Bros. The early animators, now legends in their field, toiled night and day trying to perfect the craft that we now take for granted . Perhaps the most far-reaching principle they gave us, was that of 'Squash and Stretch'.

Classical animation texts usually introduce squash and stretch by using the famous bouncing ball example. At it's core, squash and stretch is the phenomena whereby bodies in motion undergo deformations. Early uses of squash and stretch resulted in cartoon characters who's forms appear to be composed of a gelatinous substance. When one describes an animation as being 'cartoony' they are usually referring to the liberal use of squash and stretch. Exaggerated deformations can be used to convey the feeling of weight and substance. Proper use of squash and stretch is fundamental to believable animation.

This might lead one to believe that squash and stretch should be reserved for cartoon characters only. In a strict sense, this is true, but squash and stretch has come to mean something much more all-encompassing over the years. Squash and stretch can refer to not only the wobbly, squishy classical animations, but also proper muscle and skin deformations on modern digital characters. Ultra realistic creature rigs need to maintain volume throughout extreme deformations. This is the modern twist on squash and stretch.

Of all the animation principles that have been ported into the digital age, squash and stretch is perhaps the most difficult to replicate inside a computer. The reasons for this are obvious. Digital animation is, by nature, rooted in harsh, square edges and polygons. Not the sort of materials conducive to squishy motions. That being said, there are many techniques that have been honed over the years to help digital creatures squash and stretch. This chapter will attempt to show you how to add an organic, squash and stretch, element to your character rigs.
Squash and stretch can take many forms inside Maya. In addition to adding deformers to your creature's mesh (to make it bulge and stretch), you may also find it useful to have your creature's bones capable of stretching beyond their initial length. This chapter will introduce you to several techniques you can use to give your creatures that special organic quality. Specifically, we will be covering the following topics:

- Using a MEL expression to make a stretchy joint chain
- Using utility nodes to make a stretchy joint chain.
- Using cgTkStretchyIk.mel to setup stretchy limbs.
- Using blendshapes to give a creature 'cartoony' limbs.
- Rigging a stretchy spine.
- Basic vector math
- MEL scripting. The creation of cgTkStretchyIk.mel from A to Z.

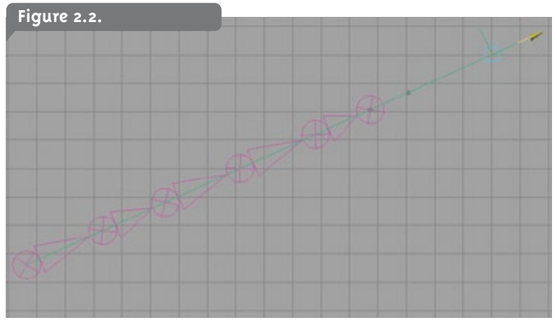
In addition to covering these specifics, this chapter will act as an introduction to one of Maya's lesser used features, utility nodes.

Exercise 2.1

Using a MEL Expression to Make a Stretchy Joint Chain

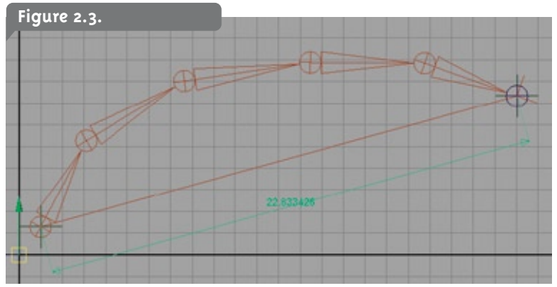
Getting your creature rigs to behave in a 'cartoony' manner involves the combination of many different techniques. Giving your creature's limbs the ability to extend beyond their natural length can give the impression of elasticity. Cartoon characters can really benefit from a rig that features limbs that allow for some elasticity.

Figure 2.2.



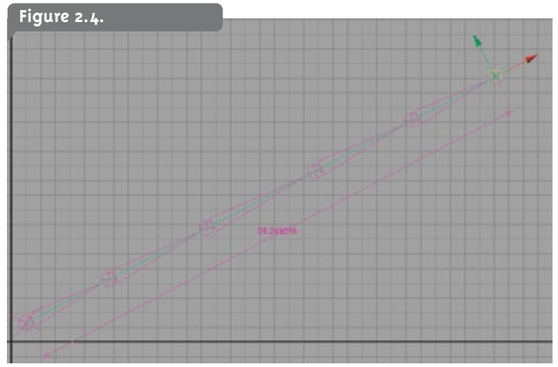
The majority of squash and stretch techniques involve the clever combination of different deformation effects. This exercise will concentrate on changing the way Maya's joints behave in an ik chain. By default, a chain of joints controlled with ik will extend to reach the ik handle until the handle moves beyond the length of the chain. At this point, the ik handle is separated from the joints FIGURE 2.2. The joints will point out in a straight line, in an attempt to reach the ik handle. This exercise will demonstrate how to create a setup whereby the joints will stretch beyond their normal length to reach an ik handle.

Figure 2.3.



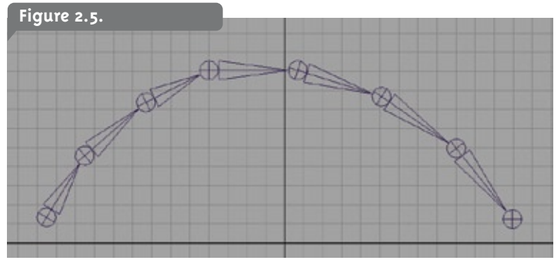
This technique uses a distance dimension tool to calculate the distance (in scene units) from the base joint in an ik chain to the ik handle FIGURE 2.3. As the ik handle is moved about, this distance will change. When the ik handle is moved beyond the length of the chain, the expression will drive the scaleX of the joints to make them stretch FIGURE 2.4.

Figure 2.4.



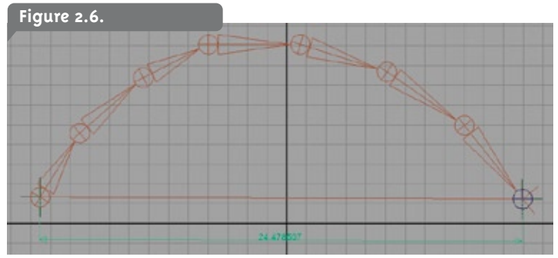
1. Open exercise2.1_Start.mb. This scene file contains a joint chain in the shape of an arch FIGURE 2.5 . Drop into the front viewport and create a single chain ik solver (default tool options) from the first to the last joint.

Figure 2.5.



2. In order to calculate the distance from the base joint to the ik handle, we will use a distance dimension node. Select `Create > Measure Tools > Distance Tool`. Hold down the 'V' key to point snap, then click once on the base joint (joint1), and again on the tip joint (joint8). This will result in the creation of two locators with a visual representation of the distance between them FIGURE 2.6. For our joint chain, this distance is 24.478507 centimeters. It is important that you remember this initial distance value.

Figure 2.6.



3. Before we begin the expression, we need to determine at what distance we want the chain to begin to stretch. This 'stretching point' occurs when the joints in the chain have rotated into a straight line in an attempt to reach the ik handle. Parent each measure tool locator under it's respective node (base joint or ik handle)FIGURE 2.7 . Now, as you move your ik handle, the distance tool will update with a new length. Move the ik handle along the positive X axis until the joint chain is straight FIGURE 2.8 . I measured a distance value of 30.6 centimeters. Undo back to before you moved the ik handle.

Figure 2.7.

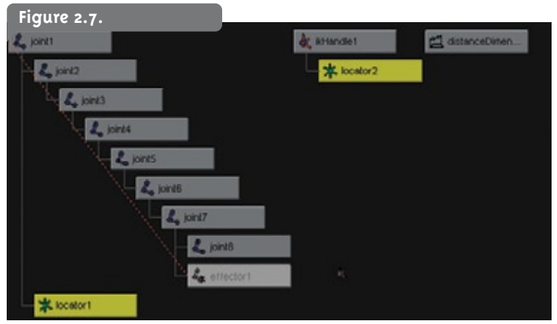


Figure 2.8.

4. Open exercise2.1_Middle.mb to see what you should have so far. We are done with the setup phase and are now ready to write the expression that will cause the chain to stretch. In order to simulate the stretching, we are going to drive the scaleX of each joint by a factor of the distance between the base joint and the IK handle. When the IK handle is moved beyond the length we determined earlier to be the maximum (30.6cm), the joints will scale along the X axis. Select the first joint. In the channel box, right click on the 'scaleX' attribute and choose 'Expressions...' to open the expression editor.

5. To start with, we need the expression to scale the joints only when the maximum distance has been exceeded by the IK handle. Let's use a conditional statement that looks at the distance dimension node's 'distance' attribute and compares that to the maximum distance (30.6cm).

```
if (distanceDimensionShape1.distance >= 30.6)
{
        //Make the joints scale (stretch)
} else
{
        //Make the scale attribute = 1 (no stretch)
}
```

6. The problem now is how do we determine how much to scale the joints so that they reach the IK handle. This can simply be done by normalizing the measured distance to a value of 1 using the following formula:

```
scale = distance / maximum distance
```

For example, if the IK handle is moved to a distance of 60 centimeters away, the scale factor will be equal to 1.96 (60/30.6 = 1.96) or roughly twice it's initial length.

7. Add the following lines of code into the 'true' section of the conditional statement:

```
//Make the joints scale (stretch)
        joint1.scaleX = (distanceDimensionShape1.distance/30.6);
        joint2.scaleX = (distanceDimensionShape1.distance/30.6);
        joint3.scaleX = (distanceDimensionShape1.distance/30.6);
        joint4.scaleX = (distanceDimensionShape1.distance/30.6);
        joint5.scaleX = (distanceDimensionShape1.distance/30.6);
        joint6.scaleX = (distanceDimensionShape1.distance/30.6);
        joint7.scaleX = (distanceDimensionShape1.distance/30.6);
```

8. The chain will now stretch properly beyond it's natural length. You may notice though, that it will not behave as expected when the distance is less than it's initial distance (24.47...). To correct this, let's explicitly tell the chain to scale by a factor of '1' when the IK handle is not stretching. Put the following code into the 'else' part of the conditional:

```
        joint1.scaleX = 1;
        joint2.scaleX = 1;
        joint3.scaleX = 1;
        joint4.scaleX = 1;
        joint5.scaleX = 1;
        joint6.scaleX = 1;
        joint7.scaleX = 1;
```

9. This is the finished expression:

```
if (distanceDimensionShape1.distance >= 30.6)
{
        //Make the joints scale (stretch)
        joint1.scaleX = (distanceDimensionShape1.distance/30.6);
        joint2.scaleX = (distanceDimensionShape1.distance/30.6);
        joint3.scaleX = (distanceDimensionShape1.distance/30.6);
        joint4.scaleX = (distanceDimensionShape1.distance/30.6);
        joint5.scaleX = (distanceDimensionShape1.distance/30.6);
        joint6.scaleX = (distanceDimensionShape1.distance/30.6);
        joint7.scaleX = (distanceDimensionShape1.distance/30.6);
} else
{
        //Make the joint's scale attribute = 1 (no stretch)
        joint1.scaleX = 1;
        joint2.scaleX = 1;
        joint3.scaleX = 1;
        joint4.scaleX = 1;
        joint5.scaleX = 1;
        joint6.scaleX = 1;
        joint7.scaleX = 1;
}
```

10. Hit the 'edit' button in the expression editor, then select the IK handle and move it around. The chain should stretch when it is pulled beyond it's initial length and behave as normal otherwise. Open the exercise2.1_Finished.mb file to see the final setup.

Exercise 2.2

Using Utility Nodes to Make a Stretchy Joint Chain

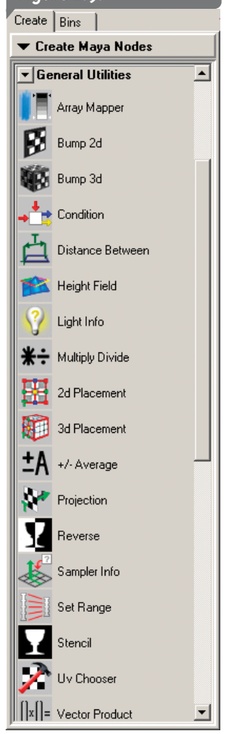
Utility nodes are Maya dependency nodes that take input data, perform some operation on it, and produce output data that can then be plugged into downstream nodes. This operation can range from simple addition/subtract to more advanced HSV to RGB color conversions. Utility nodes can be created using MEL commands or by using the hypershade's 'Create' menu FIGURE 2.9. A complete list of all the utility nodes can be found in Maya's documentation.

This exercise will act as an introduction to using Maya's utility nodes. Utility nodes can quite easily mimic the behavior of a simple expression. Their real advantage over expressions is their node based nature. A utility node is completely integrated within the dependency graph and does not need to 'fire' like an expression. This makes utility nodes much faster to evaluate.

While they are most commonly used in Maya's hypershade for developing shader networks, utility nodes have many other uses. In this exercise, we are going to replicate the exact same setup from exercise 2.1 but using utility nodes rather than an expression. We are also going to replace the distance dimension tool with a much less intrusive distance between node.

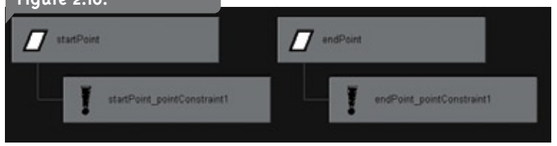
I recommend reading exercise 2.1 before continuing. This will give you a better understanding of the theory behind our technique.

Figure 2.9.



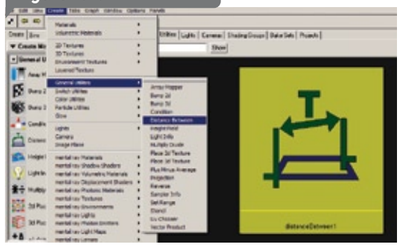
1. Open exercise2.2_Start.mb. This file contains a joint chain in the shape of an arch. Create a single chain ik handle from the base to the tip.
2. In exercise 2.1, we used a measure tool to calculate the distance between the base joint and the ik handle. This time, we are going to use the 'distance between' utility node along with some empty groups to do the calculations. The distance between node needs two vector inputs (cartesian coordinates). The problem then, is how do we find the world space coordinates of the base joint and the ik handle so that we can plug them into the distance node? I chose to use two empty group nodes, each point constrained to their respective points. Click in an empty space to clear your selection, then press `ctrl+g`. This will create an empty group node FIGURE 2.10 . Point snap the group node to the base joint. With the group node in position, point constrain it to the joint. Rename this group, 'startPoint'. Create another group node, name it 'endPoint' then point snap and constrain it to the ik handle.

Figure 2.10.



3. Because we used point constraints (instead of parenting), the group node's translates will house the world space coordinates that we want. Knowing this, we are now ready to plug them into a distance between utility node. Open the hypershade. Create a distance between node by selecting `Create > General Utilities > Distance Between` FIGURE 2.11. In the work area of the hypershade, select the newly created node. Open the connection editor (`Window > General Editors > Connection Editor`) and load the utility node into the right column by hitting the 'Reload Right' button. Select the 'startPoint' group node in the hypergraph and load this into the left column. Click on the 'translate' attribute in the left column (group node) and then click on the 'point1' attribute in the right column (distance node). This will make a direct connection between these two attributes. Load the 'endPoint' group and connect it's translate into 'point2' on the utility node. The distance node will now output the distance between these two group nodes.

Figure 2.11.



4. Right click on the distance node in the hypershade. Select 'Graph Network' to show the input connections FIGURE 2.12. If you remember from exercise 2.1, we have replicated the measure tool using utility nodes, and now we need to mimic the expression. Recall that the expression looked at the distance between the IK handle and the base joint (distance A), and compared that to the total length of the chain (distance B). If distance A is greater than distance B, the chain must stretch. Otherwise, the joint's should have a scaleX value of '1' (no stretch). We can reproduce this behavior using a condition utility node. Create a condition utility node and graph it with the distance node in the work area of the hypershade FIGURE 2.13.

Figure 2.12.

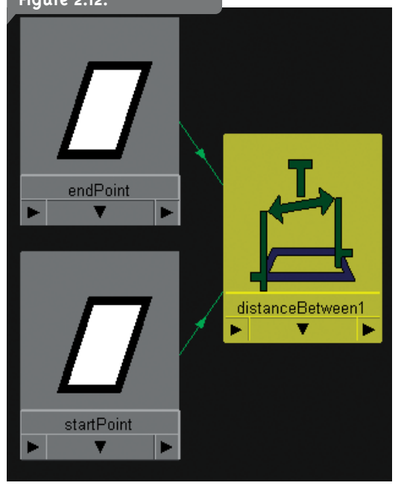
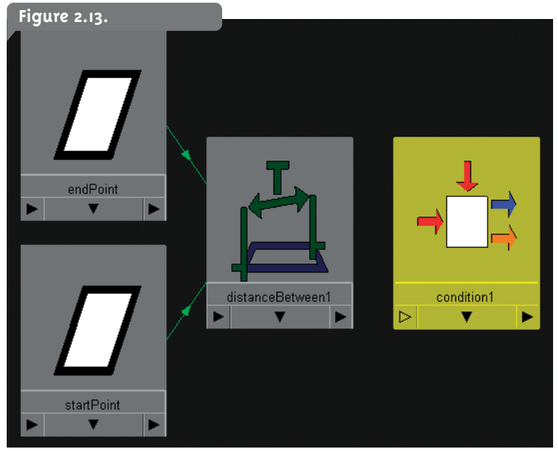
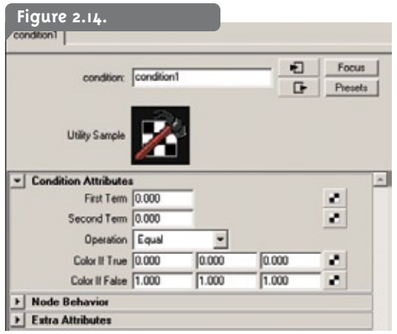


Figure 2.13.



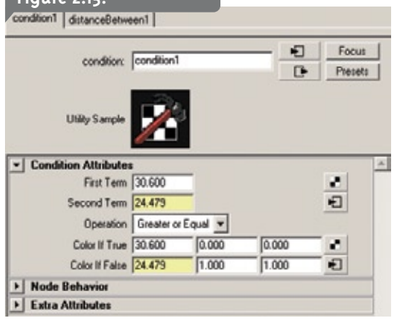
5. Select the condition node and open the attribute editor FIGURE 2.14. This node takes two float values, compares the two using simple greater-than/less-than operations, and depending on the outcome (true/false), it returns one of two vectors. Connect the 'distance' output from the distance node into the 'secondTerm' attribute on the condition node. We need the first term to equal the total length of the joint chain. This way, we can compare the two to find if the chain needs to stretch. To find the length of the chain, move the ik handle just until the joints are in a straight line. Open the attribute editor and copy the value in the second term into the first term. For this scene file, I used a value of 30.6. Now undo back to before you moved the IK handle.

Figure 2.14.



6. We have a first term and a second term (distance A and distance B respectively). Let's compare the two using the 'Greater or Equal' operation. In the case where the condition evaluates to true, let's have it return the total length of the chain (30.6). Set the 'Color if True' at R attribute to 30.6. In the case where the condition evaluates to false, we want to return the distance from the distance node. Connect the distance node's 'distance' to the 'colorIfFalseR' attribute on the condition node. Your condition node should look like FIGURE 2.15 in the attribute editor.

Figure 2.15.



7. Open exercise2.2_Middle.mb to see where we are so far. From here, we need to add one last utility node that will calculate the scale factor for our joints. Create a 'Multiply Divide' node and graph it with the condition node FIGURE 2.16. Connect the 'outColorR' from the condition node, into the 'input1X' of the multiply divide node. Set the input2X attribute equal to our chain length (30.6). Change the operation from 'Multiply' to 'Divide'.

The multiply divide node should now look like FIGURE 2.17 in the attribute editor.

Figure 2.16.

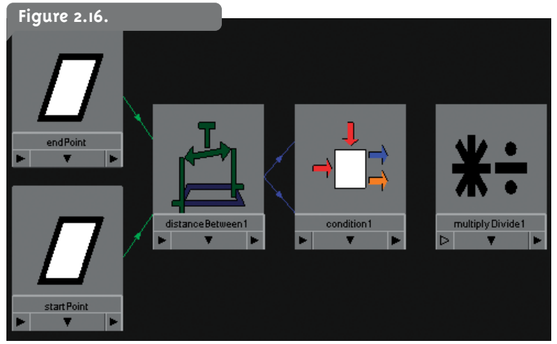
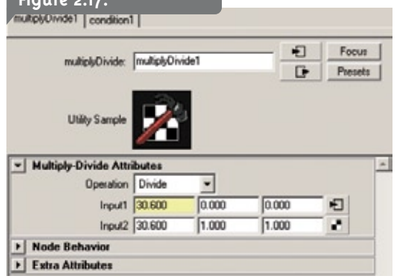
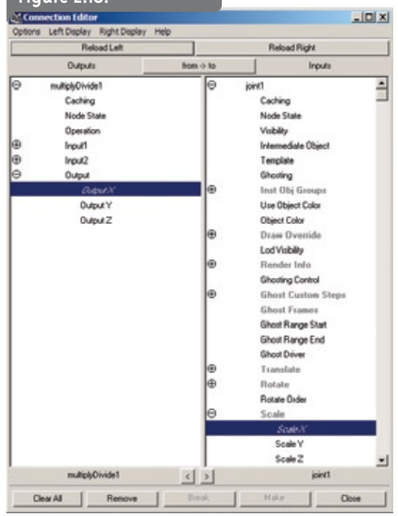


Figure 2.17.



8. The output of the multiply divide node must now be connected to the scaleX of each joint in the chain. Open the connection editor, select 'joint1' and load it into the right column. Load the multiply divide node into the left column. Connect the 'Output X' attribute from the node into the 'Scale X' of the joint. Do this for each joint in the chain FIGURE 2.18.

Figure 2.18.



9. We are now done with this setup. Try moving the ik handle around to see the joints stretch as the ik handle is moved beyond the length of the chain.

This exercise has introduced you to three very useful utility nodes. Other nodes that you may find useful are the plus/minus/average, reverse and vector product nodes. It is not uncommon for beginners to find utility nodes slightly confusing. Personally, I find them much less intuitive than writing an expression. For this reason, I usually write an expression first, then setup the utility nodes to mimic the MEL code. Whatever you do, it is good practice to use expressions sparingly.

While going through exercise 2.1 and 2.2, you may have noticed that the setup procedures are quite straightforward. Easy setups are nice, but imagine if you were responsible for setting up twenty creatures with multiple stretchy limbs. The tedium of creating and connecting all those utility nodes would be enough to drive any setup artist crazy. Once again, MEL scripting will save the day.

The next exercise will show you how to use cgTkStretchyIk.mel to automatically setup a node-based stretchy ik chain. In addition to setting up the limbs on a creature, we will also be using some custom blendshapes to simulate the maintenance of volume as the limbs undergo stretching.

Exercise 2.3 Using cgTkStretchyIk.mel to Setup a Creature's Limbs

Before starting this exercise, please ensure that the cgTkStretchyIk.mel script file is installed into your Maya scripts directory. This MEL script will create a node-based stretchy joint chain, like exercise 2.2. It works on both single chain and rotate plane ik handles. A complete line-by-line breakdown of the script is included at the end of this chapter to help inspire you to start writing your own automation scripts.

In addition to learning how to use this script, this exercise will demonstrate how to make your creature's limbs maintain volume while undergoing stretch. To do so, we are going to utilize some blendshapes that make the creature's limbs skinny. These blendshapes will then be driven automatically by the amount that each limb is stretching.

1. Open exercise2.4_Start.mb. This scene file contains a bizarre alien creature by the name of Longlee Platomas FIGURE 2.19. It has been decided that his arms and legs should be capable of stretching to meet their IK handles. Try moving his COG controller (big white one on his back) to observe the way his non-stretchy limbs behave FIGURE 2.20. You should see four IK handles at the tip of each of his limbs. These need to be made stretchy.

Figure 2.19.

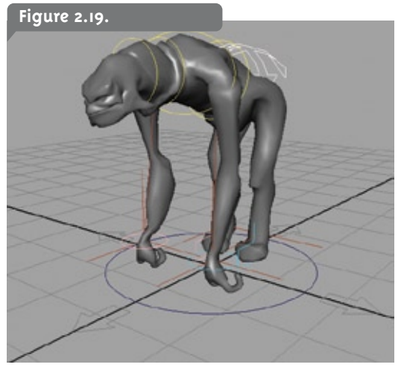
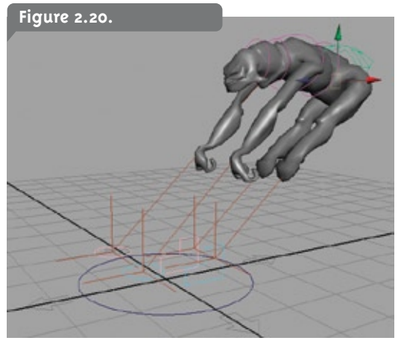


Figure 2.20.



2. With the cgTkStretchyIk.mel script installed, type 'cgTkStretchyIk;' into the command line and execute. This should bring up a small window with the 'Make Stretchy' button FIGURE 2.21. The script requires the user to select an IK handle (rotate plane or single chain only) and then hit the 'Make Stretchy' button. Do this for all four of the IK handles on Longlee Platomas, in this order:
lLegIk
rLegIk
lArmIk
rArmIk

Figure 2.21.

3. If you are using this script on your own characters, ensure that each IK handle has a unique name. The script will error out if there are multiple IK handles with the same names. To test and see if the script has setup the stretchiness properly, grab the COG controller and move it around FIGURE 2.22. Longlee's limbs should remain planted on the ground regardless of how far the COG controller is translated. Open the hypershade and click on the 'Utilities' tab. You should see all of the utility nodes that were created to setup the stretchy limbs FIGURE 2.23. This script uses the exact same procedure from exercise 2.2.

Figure 2.22.

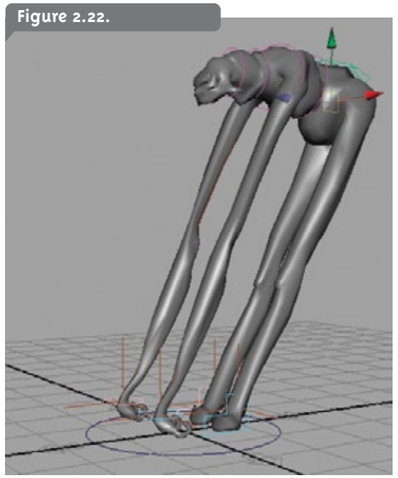
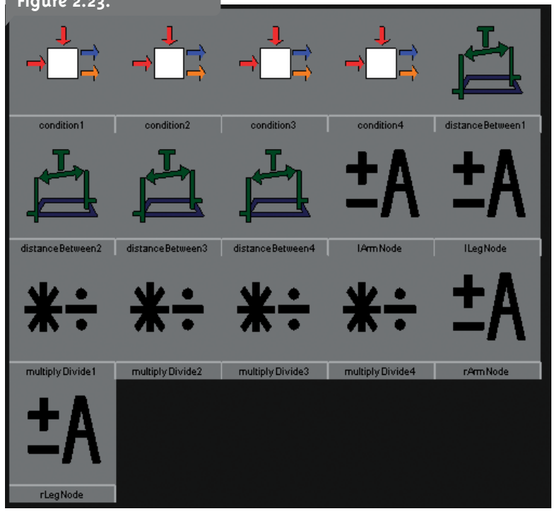
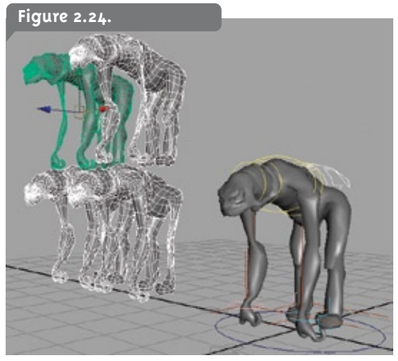


Figure 2.23.



4. The joint chains are now behaving properly, but you may notice that Longlee Platomas' limbs retain their original girth even when undergoing extreme stretching. If you want a physical demonstration of what should actually be happening, you needn't look any further than an elastic band. When stretched, an elastic band will get noticeably thinner because it must cover more space. To get Longlee's limbs to stretch thin, we will use some simple blendshapes. Unhide the layer named 'Blendshapes'. You should see four copies of the mesh, each with one of the limbs modeled thin FIGURE 2.24.

Figure 2.24.



5. Open exercise2.4_Middle.mb to see our progress so far. In order to see the exact effect of each blendshape on the mesh, let's apply them and test. Select all four duplicate meshes, then shift select the rigged mesh. In the animation menu set, select `Deform > Create Blendshape`. Go to the options box and set the blendshape node name to 'stretchShapes', then hit 'Create' FIGURE 2.25. Now hide the 'Blendshape' layer and select the rigged mesh. On the inputs of 'Mesh', you should see the stretchShapes node sitting above the skinClusteri node. If you select the stretchShapes name, the channel box will display all the target shapes and their values FIGURE 2.26.

Figure 2.25.

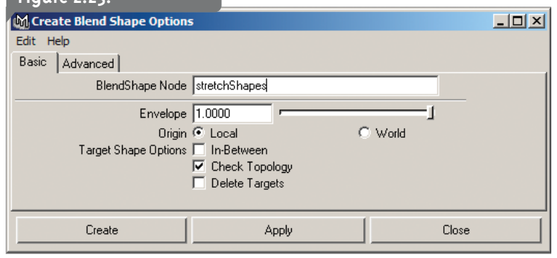
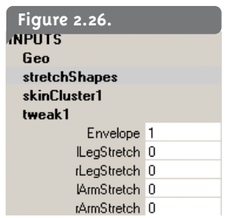
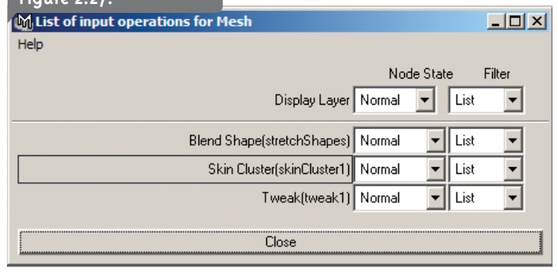


Figure 2.26.



6. Select the lLegStretch attribute on the stretchShapes input and middle mouse drag in the viewport. Observe the thinning effect it has on Longlee's left leg. Try out each blendshape to see their effects. While the blendshapes may appear to be working correctly, we have encountered a deformation-order problem. If Longlee is not in his default pose, the blendshapes will transform him back to his default pose. This, of course, is not what we want. When we applied the blendshape, it was after having applied the mesh's skinCluster or smooth skinning. As a result, the blendshape is at the end of the deformation order for the mesh. We must edit this order such that the skinCluster is before the blendshape node. Select the mesh, right click to bring up the marking menu. Select `Inputs > All Inputs...` from the marking menu. This will bring up the list of inputs for our mesh FIGURE 2.27. Notice the blendshape input is at the top of the list. Middle mouse drag the blendshape node on top of the skin cluster. This will switch their order so that it behaves as we wanted. To test it out, move the COG up, then try the blendshapes. They should deform as expected.

Figure 2.27.



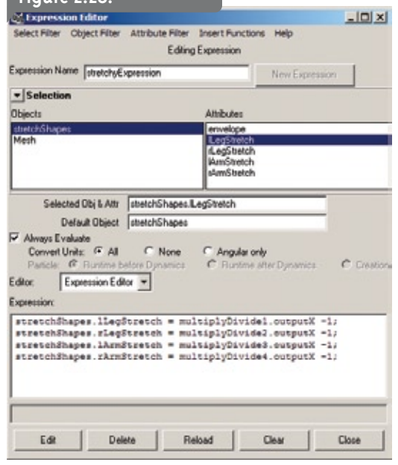
7. Now that our blendshapes are setup correctly, we need to connect them so that they are automatically driven by the amount that the limb is stretched. You can use whatever method you want (set driven key, utility nodes etc...) but I chose to use an expression for our demonstration purposes. In order to get these blendshapes to stretch when they should, we are going to hook them up to the same utility nodes that the script created to make the joint chain stretch. Select your mesh, in the channel box, open the stretchShapes input and right click on one of the blendshapes. Choose 'Expressions' from the pop-up menu to open the expression editor.

8. Name the expression 'stretchyExpression'. In the text field, enter the following MEL code:

```
stretchShapes.lLegStretch = multiplyDivide1.outputX -1;
stretchShapes.rLegStretch = multiplyDivide2.outputX -1;
stretchShapes.lArmStretch = multiplyDivide3.outputX -1;
stretchShapes.rArmStretch = multiplyDivide4.outputX -1;
```

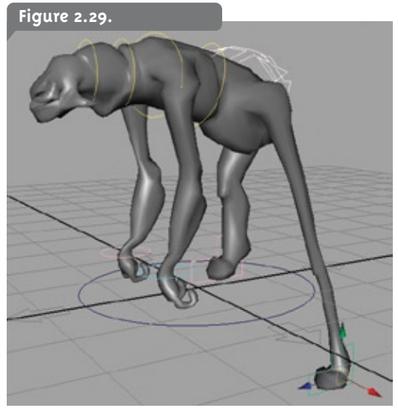
Press the 'Create' button to make the expression FIGURE 2.28.

Figure 2.28.



9. If you setup your stretchy ik handles in the order specified in step two, the expression should work fine. Test it out by tugging on his individual limb controller curves. His limbs should stretch thin as they are pulled further from his body FIGURE 2.29. If you notice that pulling one limb causes a different limb to thin up, you have likely created your stretchy chains in a different order. Determine which multiplyDivide node is affecting which limb and then update the expression to solve this problem, should it arise.

Figure 2.29.



10. You may also wish for your creatures limbs to stretch thin at a different rate. You can adjust the rate by altering the value added at the end of each line in the expression. For example, the following edit to the expression will cause the limb to stretch at a slower rate.

```
stretchShapes.lLegStretch = multiplyDivide1.outputX -1.5;
```

All I did was change the '-1' to '-1.5'. Try different values to see their affect on the limb's rate of thinning.

11. Open exercise2.3_Finished.mb to see the final rig. Longlee's limbs are now fully capable of stretching beyond their initial length and they realistically preserve volume by getting thinner as they stretch. For further practice, you may want to try swapping out the expression in favor of a couple utility nodes with direct connections. This could quite easily be done using the plus/minus/average node.

In addition to using stretchy limbs, your digital cartoon creatures may benefit from having a spine setup that is capable of stretching. The problem with stretchy spines is that you usually need more control over the orientation of the chain joints than what is available using the stretchy IK technique.

In addition to being useful for cartoon effects, a stretchy spine can work for realistic characters as well. When used sparingly, a stretchy spine can give even realistic creatures a more organic quality.

The basic procedure involves using a spline ik solver to control the spine joints. The spline ik curve that runs through our spine will control the orientation of the joints. The CV's on this curve are clustered and parented to a couple of controller curves. To allow the spine joints to stretch when the clusters are moved too far, the scaleX of each joint is connected to a multiply divide node that normalizes the current length of the curve against it's initial length. Because clusters do not provide a twisting control on the spline IK chain, we must setup additional control via an expression.

The resulting rig will allow for quite a bit of control while maintaining an intuitive familiarity for those who are used to pure FK spine setups.

Exercise 2.4 Rigging a Stretchy IK/FK Spine

For this exercise, you will be guided through the step-by-step process of setting up a stretchy spine. Because spine rigs must work in conjunction with the rest of a creature's body parts, (legs, arms, head etc...) I chose to do this demonstration on a human skeleton model. In addition to the arm and leg bones, this model includes an anatomically correct model of a human spine. There are twenty-four vertebrae in a human skeleton made up of five lumbar, twelve thoracic and seven cervical.

1. Open exercise2.4_Start.mb. This scene file contains a model of the human skeleton's limbs and spine FIGURE 2.30. This model is made up of several polygonal objects that are either smooth bound or parent constrained to their respective Maya joints. While you are completing this exercise, you may find it useful to turn off the display layers to hide the meshes whenever they clutter your view. If you turn off both display layers right now, you should see a joint chain running up the spine FIGURE 2.31. Before setting up these joints, we must ensure that they are oriented correctly. Select the base joint (l5) and execute the following MEL command:

```
select -hi;
```

This will select the entire hierarchy of joints below the currently selected one. I keep this handy MEL command in a button on my shelf.

Figure 2.30.

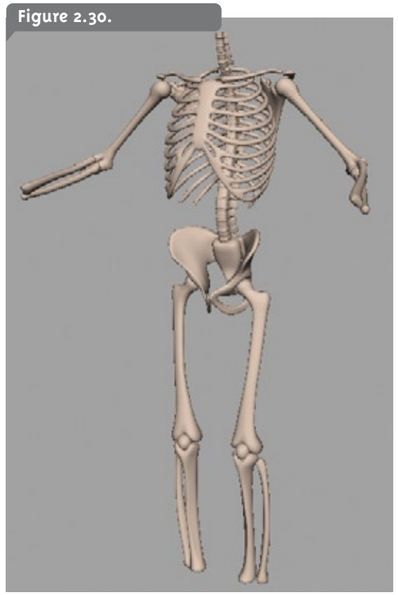


Figure 2.31.

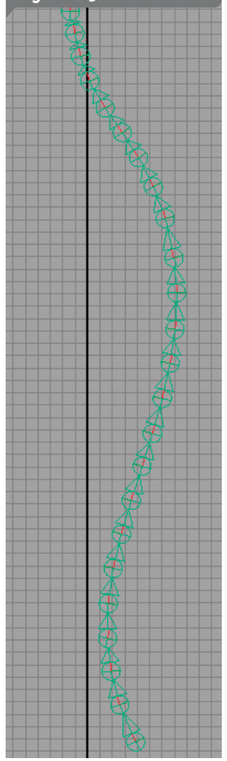
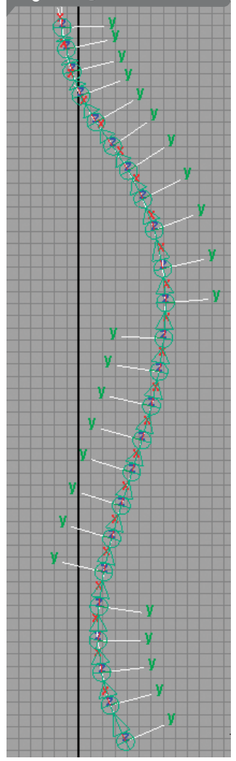


Figure 2.32.

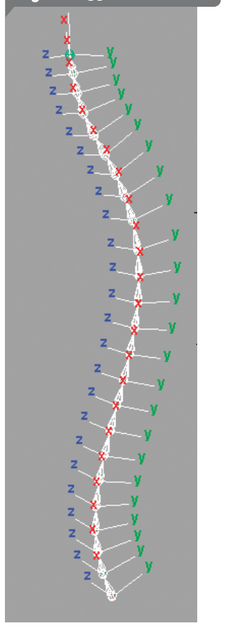


2. With all the spine joints selected, we need to check their local rotation axis to ensure that they are pointing the same way. Execute `Display > Component Display > Local Rotation Axes` to display the rotation axes of each joint FIGURE 2.32. If you look at the joint chain, you can plainly see that the entire middle section of the spine has it's axis offset. To correct this, we need to select the rotation axis itself. Go into component mode and click on the question mark icon in the object type bar. You can now select and manipulate the local rotation axis. In order to ensure an exact orientation, we are going to use a MEL command to rotate the axis into alignment rather than rotating it manually in the viewport. If you look at the axis, the positive Z direction is facing to the left when it should be facing to the right. To flip it 180 degrees, select the axis and execute the following MEL command:

```
rotate -r -os 180 0 0;
```

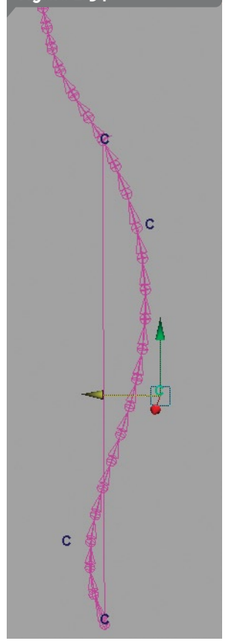
3. Once all of the rotation axis are properly aligned (FIGURE 2.33), you can turn off the display of the local rotation axis. The scene file, exercise2.4_MiddleA.mb, contains the properly aligned spine joints. Now we need use the spline ik solver to control the orientation of these joint. Select `Skeleton > Ik Spline Handle` and go to the options box. Reset the tool and turn 'Auto Parent Curve' to 'off', and the number of spans to '2'. Click once on the bottom joint (l5) and once on the joint named 'c7' (about two thirds up the chain). This will result in the creation of a curve that runs through the spine joints as well as a spline ik handle.

Figure 2.33.



4. In order to control the curve, we need some way of manipulating the control vertices. Fortunately, Maya has a deformer that will do the trick. You may find it handy to hide the joints before clustering the CV's. Select each CV along the curve that runs through the spine (there should be five) and execute `Deform > Create Cluster` for each CV. Use the default creation options for your clusters. This will result in the creation of five cluster handles FIGURE 2.34. Grab one and tug on it to see it's effect on the joint chain.

Figure 2.34.



5. In order to make the joint chain stretch, we are going to use a technique similar to that of exercise 2.2. We will find the length of the spline IK curve, then feed that into a multiply divide node where it will be normalized before being plugged into the scaleX of each joint in the spine. To find the length of a curve, Maya has a handy node called a curveInfo node. To create one for our curve, select the curve and execute:

```
arclen -ch 1;
```

6. Create a multiply divide node from the hypershade and name it 'spineStretchNode'. Select it and the spline IK curve before hitting the input/output connections button in the hypergraph. This will display both of their dependency graphs. Locate the curveInfo node in the hypergraph and drag it near the multiply/divide utility node to make it easier to make the necessary connections FIGURE 2.35. Connect the 'arc length' from the curve info node into the 'input1X' of the multiply/divide node. Double click the utility node to open the attribute editor. Copy the value from input1X into input2X (this should be 48.463). Change the operation from 'Multiply' to 'Divide' FIGURE 2.36.

Figure 2.35.

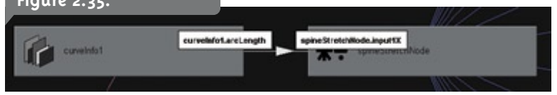
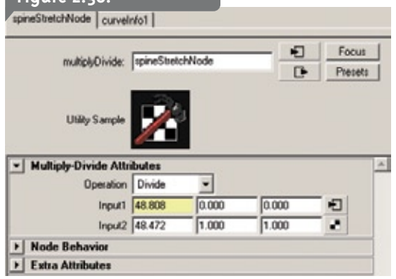
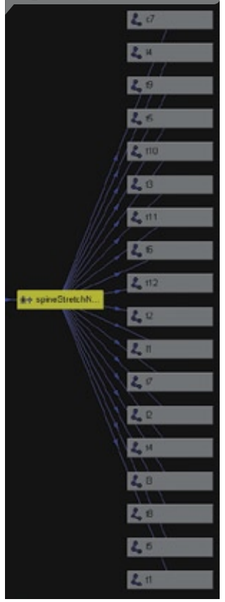


Figure 2.36.



7. The multiply/Divide node's output is now ready to drive the scaleX of each joint. Load the utility node into the left column of the connection editor and load the first spine joint into the right column. Connect the 'OutputX' from the utility node to the 'ScaleX' of each joint in the spine from joint 'l5' to joint 'l1' FIGURE 2.37.

Figure 2.37.



8. You can now tug on a cluster to observe how the chain will stretch to accommodate the curve's length FIGURE 2.38 . Open exercise2.4_MiddleB.mb to see the rig thus far. Rather than use these

clusters, we will make some controller curves for the animator to use. Create a circle and point snap it to the IK handle. Scale and position this curve however you want, then freeze the transforms on it. Name this 'shoulderControl'. Parent the top two clusters to this curve. Create and position another controller curve (at the l5 joint) , this time name it 'hipsController'. Parent the bottom two clusters to this controller FIGURE 2.39.

Figure 2.38.

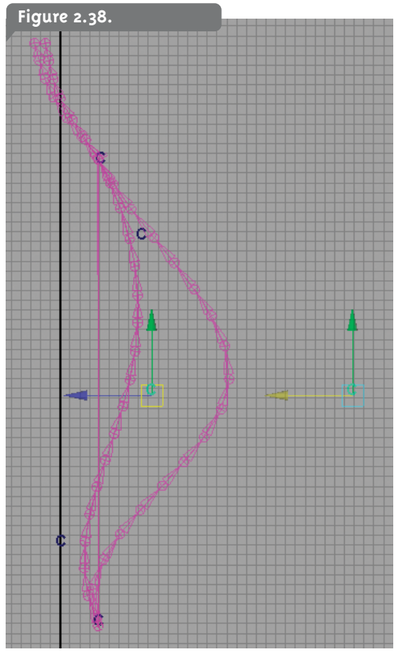
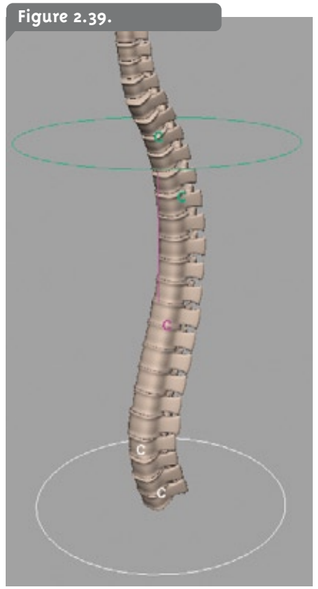


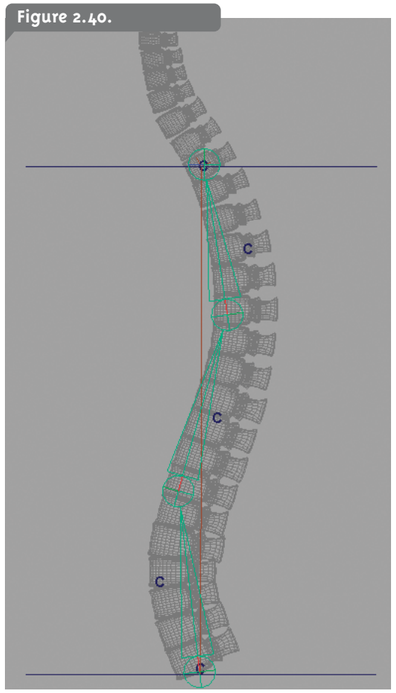
Figure 2.39.



9. You may notice that the last (middle) cluster has not been parented. This cluster can simply be parent constrained to both the hipControl and shoulderControl curves. Select the shoulderControl curve, shift select the middle cluster and execute `Constrain > Parent`. Choose the default options in the parent constraint options box and then hit 'Create'. Do this for the hipControl curve as well. Open exercise2.4_MiddleC.mb to see the progress this far.

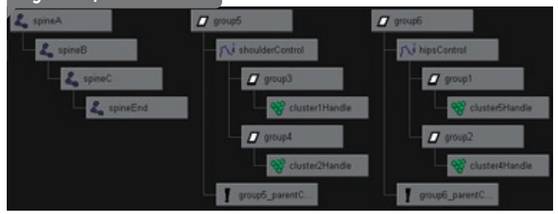
10. The stretchy IK setup is now finished. To give this spine more control, we are going to give it some twisting ability as well as the ability to be posed in FK fashion. Drop into a side viewport and create three bones (four joints) along the length of the spine FIGURE 2.40. These joints should be placed along the length of spine where you want to have FK control. Name these joints, spineA, spineB, spineC and spineEnd.

Figure 2.40.



11. To get the spine to react in an FK manner, we need to constrain the hip and shoulder controls to this newly created joint chain. Group the shoulderControl curve to itself and center the pivot. Then parent constrain the new group node to 'spineEnd'. Group the hip controller to itself and parent constrain it to 'spineA' FIGURE 2.41. Try positioning the FK chain to see how the spine behaves. Open exercise2.4_MiddleD.mb to see the rig at this point.

Figure 2.41.

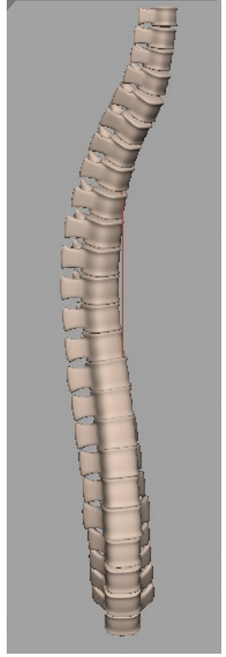


12. The spine is fairly pose-able, but it lacks the ability to twist properly. To twist the spine, we are going to take advantage of the 'twist' attribute on the spline IK handle. Select the IK handle and middle mouse drag on the twist attribute to see it's affect FIGURE 2.42 . We can set this up to be automatically driven by the rotation of the FK joint and the shoulder control by uses a simple MEL expression. With the IK handle selected, find the 'twist' attribute in the channel box and right click on it. Select 'Expressions...' from the pop-up menu. Name the expression 'spineTwist'. In the expression editor's text field, enter:

```
ikHandle1.twist = (spineA.rotateX + spineB.rotateX +
spineC.rotateX + shoulderControl.rotateY);
```

Hit 'Create'. Try rotating the FK controls to see the spine twist.

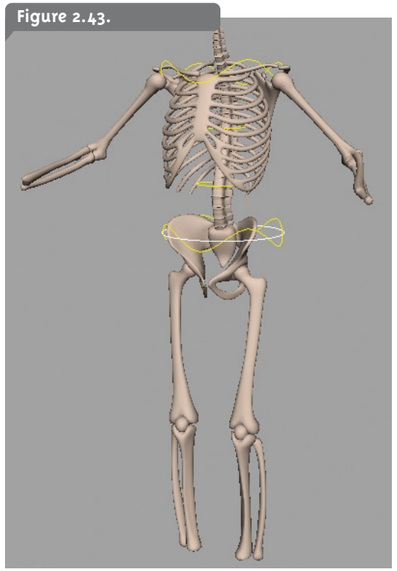
Figure 2.42.



13. The last thing to setup is a control that will move the entire back. Create another curve, position it at the hip and name it 'COG' for 'Center Of Gravity'. Parent the spine hierarchy under this controller. Parent constrain the 'spineA' control joint to the COG.

14. Feel free to clean up the hypergraph by grouping the spine nodes appropriately. Also, make sure to lock and hide any unused channels in your control objects. To make the fk chain easier to manipulate, you could attach some curves to them as well. Open exercise2.4_Finished.mb to see the final rig FIGURE 2.43.

Figure 2.43.



The finished scene file from exercise 2.4 includes a stretchy spine, legs and arms. If you grab the COG controller and translate it around, you can observe a completely stretchy bipedal rig. The arms and legs were quickly setup using cgTkStretchyIk.mel and the spine was setup using the process outlined in exercise 2.4.

■ MEL Scripting

The last section of this chapter is going to detour around the concept of squash and stretch to focus on the creation of the cgTkStretchyIk.mel script file. If you have read the coverage of the script from chapter one, you should have no problem following along with this one. Novice MEL scripters will benefit from watching the MEL introduction video from the DVD before continuing with this exercise.

The script file we are going to dissect was created to replicate, exactly, the process from exercise 2.2. Please read exercise 2.2 and ensure that you have a solid understanding of the process we are trying to emulate before continuing.

■ Mathematics and MEL

While you can certainly create some amazing MEL scripts without ever utilizing any math whatsoever, there will inevitably come a time when knowledge of basic math principles will help you to solve a scripting problem.

In addition to the trigonometry you were taught in high school, basic vector maths will come in handy now and then. I have a couple of 3d math books, geared towards programmers, and they have proven indispensable. If you are like me (and never pursued much math beyond college), I would recommend picking up a 3d math text or even attending a course at your local college. You never know where you'll find a use for it.

In the case of cgTkStretchyIk.mel, I found that I needed a way to compute the length of any joint chain of arbitrary shape. If you recall from exercise 2.2, in order to setup the multiply/Divide node to normalize the scale factor for our joints, we needed to measure the length of the chain when laid out flat. To do this, we simply tugged on the IK handle until the chain was straight, and then we used a distance dimension tool to measure the length. This is where the problem arises. How do you tell a script to 'tug on an IK handle until the chain is straight' ? You might be able to determine if the chain is straight by measuring angles and whatnot but a much easier, more elegant method, involves the use of the distance formula from vector mathematics. We can simply measure the distance between each joint, and add them all together to arrive at the total length of the joint chain.

■ The Distance Formula

Perhaps the most important formula in 3d math, the distance formula, will simply give you the distance between two points in space. Given point A and point B, (both A and B are vectors of the form x,y,z) we can compute the distance between them.

Before I break out the formula, it is important to understand a little about vectors and the built in MEL functions that automatically handle their computation. A vector, in mathematics, is any list of numbers. Most commonly, vectors are either two dimensional, 3d or 4d. This means they have either two, three or four numbers in their list. In MEL, the vector variable type is simply a float array with 3 indices. The MEL language has built in functions for doing various operations on 3d vectors.

```
//A float array housing three float values.
float $float[2];
//A vector variable, also houses three float values.
vector $vector;
```

The reason that MEL has native support for only 3d vectors (and not vectors of any other dimension) is because of the inherent usefulness of 3d vectors. A 3d vector can represent a point in 3d space (x,y,z). Fortunately, the distance between two points in 3d space can be quite easily calculated using vectors.

To find the distance between two vectors, (point A, point B) you must start by subtracting them. This will result in the creation of a third vector , C. Now, if we calculate the magnitude of C, we will be left with the distance from A to B. In MEL, it looks like this:

```
//Declare our two points
vector $pointA = <<5,7.6,9>>;
vector $pointB =<<1,1,1>>;

//Subtract the first vector from the second.
//This will result in the creation of another
//vector.
vector $afterSubtraction = $pointA - $pointB;

//Compute the magnitude of the new vector
float $distance = mag ($afterSubtraction);

//Print the distance
print  ("The distance from pointA to pointB is "+...
        $distance);
```

This MEL code will print out:

```
The distance from pointA to pointB is 11.11575459
```

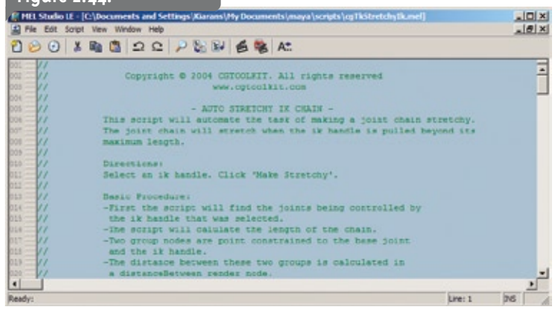
Thankfully, we don't need to know how to subtract vectors or calculate the magnitude to use the distance formula. MEL does this for us automatically. Many other programming languages require that you create a special vector class in order to do these kinds of higher-level calculations. Yay for MEL!

Keep this formula in mind because we are going to use it in the next section where I describe, in detail, how the cgTkStretchyIk.mel script works.

The Creation of cgTkStretchyIk.mel-

In the last section of chapter three I go over, in detail, the creation of the cgTkDynChain.mel script. I also describe the sort of ideal work flow for writing and sourcing your scripts. I highly recommend using Mel Studio LE which is included on the DVD in this book for developing your scripts. Install Mel Studio and open cgTkStretchyIk.mel to take a look at it FIGURE 2.44.

Figure 2.44.


MEL Studio LE - [C:\Documents and Settings\Kiaran\My Documents\maya\scripts\cgTkStretchyIk.mel]
File Edit Script View Window Help
Copyright © 2004 CGTOOLKIT. All rights reserved
www.cgtoolkit.com
- AUTO STRETCHY IK CHAIN -
This script will automate the task of making a joint chain stretchy.
The joint chain will stretch when the ik handle is pulled beyond its
maximum length.
Directions:
Select an ik handle. Click 'Make Stretchy'.
Basic Procedure:
-First the script will find the joints being controlled by
the ik handle that was selected.
-The script will calulate the length of the chain.
-Two group nodes are point constrained to the base joint
and the ik handle.
-The distance between these two groups is calculated in
a distanceBetween render node.
Ready
Line: 1

This script file follows a series of conventions with regards to the way it is organized. At the very top of the script, I have included a header. The header for this script includes all the pertinent information like how to use it and what the script actually does in the background (it's procedure).

```
//
//      Copyright © 2004 CGTOOLKIT. All rights reserved
//                                      www.cgtoolkit.com
//
//                          - AUTO STRETCHY IK CHAIN -
//     This script will automate the task of making a joint chain stretchy.
//     The joint chain will stretch when the ik handle is pulled beyond its
//     maximum length.
//
//     Directions:
//       Select an ik handle. Click 'Make Stretchy'.
//
//       Basic Procedure:
//       -First, the script will find the joints being controlled
//        by the ik handle that was selected.
//       -The script will calulate the length of the chain.
//       -Two group nodes are point constrained to the base joint
//        and the ik handle.
//       -The distance between these two groups is calculated in
//        a distanceBetween render node.
//       -The chain length is fed into a condition and multi/div
//        render node where it is divided by the distance.
//       -The result of the division is plugged into the
//        scaleX of each joint in the chain. This causes
//        the joints to stretch.
```

Always include a header in your script files. Even if you are the only one using it, they serve as a reminder of exactly what the script does. If you are as forgetful as me, this will save the day when you return to the script months down the road to change it in some way.

This script is comprised of several procedures, each separated by three lines of commented text that encompass the name of the procedure. This will make it easier to scroll through longer scripts to find a specific part.

```
////////////////////////////////////////////////////////////////////////////
//                          makeIkStretchy procedure
////////////////////////////////////////////////////////////////////////////
```

If you scroll all the way to the bottom (line 181), you will find the last procedure in the script (cgTkStretchyIk). This procedure houses the user interface, and it is what will be called when the script is executed. I made this a global procedure so that it can be called from anywhere.

```
global proc cgTkStretchyIk ()
{
        //UI goes here...
}
```

Unlike the script from chapter one, this script will call a window that will display a button for the user. To ensure that this window will delete itself if called while it is active, I have added a conditional to the top of the procedure:

```
if (`window -q -ex stretchyIkWindow`) deleteUI stretchyIkWindow;
```

If the 'window' command is queried to return the state of it's existence, we can determine whether or not to delete it before calling it again. This ensures that multiple windows will not be created each time the user calls the procedure.

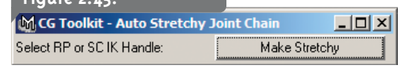
Now it is safe to create the window:

```
//Main Window
window -title "CG Toolkit - Auto Stretchy Joint Chain" ...
        -w 340 -h 50 stretchyIkWindow ;
```

I arrived at a width and height of 340 and 50 respectively after adding all of the UI elements (buttons and text). Then different values were tested to arrive at a window that fit correctly.

The interface for this script will be extremely simple. In fact, it only contains one button and some simple instructions FIGURE 2.45. For a script like this, you needn't even use a window if you do not want to. The user could simply access the functionality of the script through the use of a global procedure call. In fact, if you want to bypass the interface entirely, this script can be used by selecting an IK handle and executing 'makeIkStretchy;' from the command line. This is exactly what the button does but some people prefer a button over a written command. Regardless, the interface is comprised of a single row/column layout with some text instructions and a button.

Figure 2.45.




```
//Button Layout
rowColumnLayout -nc 2 -cw 1 175 -cw 2 150;
text "Select RP or SC IK Handle: ";
button -label "Make Stretchy" -c "makeIkStretchy";
```

The rowColumnLayout command creates a layout with two columns, as specified by the '-nc 2' flag. The '-cw 1 175' and '-cw 2 150' flags specify the width of each column in pixels. Column one is 175px and column two is 150px wide. These values provide enough space to encompass the text and the button elements.

The button labeled 'Make Stretchy' will call the 'makeIkStretchy' procedure which houses the commands necessary to setup a stretchy IK joint chain.

Lastly, the window must be called. This is done using the 'showWindow' command followed by the name of the window UI element (in this case 'stretchyIkWindow').

```
//Show Main Window Command
showWindow stretchyIkWindow;
}
```

This is the end of the user interface procedure. It is very simple, but serves it's purpose.

■ The 'makeIkStretchy' Procedure

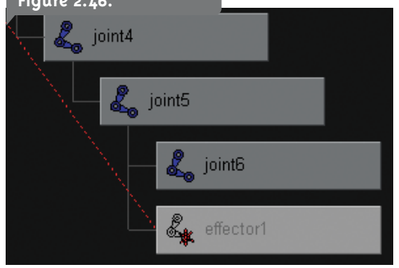
When called, (either from the button in the UI or manually) this procedure requires that the current selection include only one IK handle. Knowing the currently selected IK handle, the script is able to determine what joints need to be made stretchy. Start by declaring the global procedure, and initiating some variables to house the name of the IK handle.

```
global proc makeIkStretchy ()
{

//Store the current selection into string array.
        string $sel[] = `ls -sl`;

//Store the name of the IK handle into a string.
        string $ikHandle = $sel[0];
}
```

With $ikHandle now housing the name of the IK handle that the user wants to make stretchy, we need to determine what joints that IK han

dle is affecting. We cannot , unfortunately, query this directly, but we can query the end effector of the IK handle. Knowing the name of the end effector we can do a little bit of pick walking to find the end joint. Knowing that the end effector is always parented under the second last joint that the IK handle is controlling FIGURE 2.46, we can simply pick walk up and then down to find the end joint. In MEL, it looks like this:

Figure 2.46.




```
//Find the end joint where the ikHandle is located.
string $endJoint[];
$endJoint[0] = `eval ("ikHandle -q -endEffector " + $ikHandle)`;
select $endJoint[0];
$endJoint = `pickWalk -d up`;
$endJoint = `pickWalk -d down`;
```

Fortunately, we can find the start joint by simply querying the IK handle.

```
//Find the start joint being affected by the ik handle.
string $startJoint[];
$startJoint[0] = `eval ("ikHandle -q -startJoint " + $ikHandle)`;
```

You may notice that I used string arrays to house the start and end joint names. While each variable only needs to house one string, I was required to use an array because the 'eval' command returns a data type that can only be stored into an array. It may sound odd (and it is) but if you don not use an array, the code will not execute and Maya will return an error. It is easy to catch this type of bug because Maya will tell you that it cannot 'convert data of type string to type string[]'. Just be mindful of this caveat.

As mentioned in the earlier section on vector maths, we cannot use the same method from exercise 2.2 to determine the length of the joint chain. This is because exercise 2.2 required that the user pull the IK handle until the joint chain was straight and then measure the distance. Of course, in order to calculate the length of the joint chain using the distance formula, we must know the x,y,z coordinates of each joint along the chain. I chose to use a while loop to iterate over each joint, and find it's world space coordinates. Before the next iteration, the loop also calculates the distance between the current point and one before it. It then adds this distance to the $totalDistance variable which will house the full length of the chain when the loop is done.

To start with, we need to declare some variables and prepare the selection before entering into the while loop:

```
//Create a vector array to store the world space coordinates of the joints.
vector $jointPos[];
//Vector between two points
vector $btwPointsVector = <<0,0,0>>;
//Create a float to store the distance btw the current joint and the last one.
float $distBtwJoints = 0;
//This will store the total distance along the length of the chain.
float $totalDistance = 0;
//String variable to house current joint being queried in the while loop.
string $currentJoint = $startJoint[0];
//Counter integer used in the while loop to determine the proper index in the
//vector array.
int $counter = 0;
//Exit loop boolean
int $exitLoop = 0;

//Initial selection going into the while loop
select $startJoint;
```

With all of these out of the way, we can begin to construct the loop. The loop will start with the first joint in the IK chain. At the end of each iteration, the loop will pick walk downwards until it reaches the end joint in the chain. At this point, the $exitLoop integer will be set to '1' and thus the loop will exit.

```
//The while loop keeps going until the current joint equals the end joint.
        while ($exitLoop == 0)
        {
                //Exit loop condition
                if ($currentJoint == $endJoint[0])
                {
                        $exitLoop = 1;
                }

                //Query the world space of the current joint.
                $jointPos[$counter] = `joint -q -p -a $currentJoint`;

                //DO DISTANCE CALCULATIONS

                pickWalk -d down;
                $sel = `ls -sl`;
                $currentJoint = $sel[0];
                $counter++;
        }
```

The loop is pretty straightforward so far. To query the world space of a joint, you can simply use the 'joint' command. Notice that I am storing the world space of the current joint into a single variable ($jointPos). This will allow us to calculate the distance between this joint and the last one. Insert the following code at the line that currently says '//DO DISTANCE CALCULATIONS'.

```
if ($counter != 0)
{
        //Calulate the distance between this joint and the last.
        //First compute the vector between the two points
        $btwPointsVector = ($jointPos[$counter-1]) - ($jointPos[($counter)]);
        //Now compute the length of the vector (the magnitude)
        $distBtwJoints = mag ($btwPointsVector);
        //Add the distance onto our total
        $totalDistance = ($totalDistance + $distBtwJoints);
}
```

I've encompassed the entire distance calculation under a condition that asks if the $counter variable is equal to '0'. If the $counter variable were equal to '0', it would mean that we are going through the first iteration of the loop and it would be impossible to calculate the distance from the first joint to the one before it since there are none before it.

If the loop has passed the first joint, it can begin doing the distance calculations and tallying up the total distance. If you remember from the section about vector maths, this involves first subtracting the two points from each other, then calculating the magnitude of the resulting vector. This magnitude is the distance between the two points. That distance is added to the $totalDistance attribute which will later be used to normalize the scale factor for all the joints in the chain in order to make them stretch.

The trickiest part of this script is over. From here, I will demonstrate how to setup utility nodes (like exercise 2.2) using MEL. Recall that the distance between utility node requires two vector inputs (two points in 3d space). We could simply query the world space of the IK handle and the base joint, but the distance node needs to be fed an updated coordinate as the ik handle is moved around (and thus the distance changes). I chose to use two empty group nodes to find the world space coordinates of the base joint and IK handle. These group nodes are constrained to their relative nodes (base joint and IK handle) so that their translate channels house the world space coordinates that we need.

```
//To measure the distance from the ik handle to the start joint,
//create two empty group nodes and use there translates to
//calculate the distance using a distanceBetween render node.
string $startPoint = `group -em`;
string $endPoint = `group -em`;
$startPoint = `rename $startPoint ($ikHandle + "startPoint")`;
$endPoint = `rename $endPoint ($ikHandle + "endPoint")`;
pointConstraint -offset 0 0 0 -weight 1 $startJoint[0] $startPoint;
pointConstraint -offset 0 0 0 -weight 1 $ikHandle $endPoint;
```

The group nodes are now point constrained and ready to have their translates fed into a distance node.

```
//Create a distance between render node.
string $distanceNode = `shadingNode -asUtility distanceBetween`;
```

A distance between render node is created and it's name is stored into the $distanceNode string variable. To connect the translates, we can simply use the 'connectAttr' MEL command.

```
//Connect the translates of the point constrained grp nodes
//to the point1 and point2 inputs on the distance node.
connectAttr -f ($startPoint + ".translate") ($distanceNode + ".point1");
connectAttr -f ($endPoint + ".translate") ($distanceNode + ".point2");
```

Our distance node now has the proper connections and we are ready to use it's output. Please recall that the setup in exercise 2.2 uses a condition and a multiply/divide node as well. The condition node compares the current distance (from the IK handle to the base joint, distance A) with the total length of the chain (B). If A is greater than B, the chain must stretch and so the condition node returns the current distance. If A is less than B, the joints in the chain should have their scales set to '1' and so the condition node returns the intial length (B) which is simply divided by itself to scale the joints by a factor of '1' (no scaling).

```
//Create a condition render node.
string $conditionNode = `shadingNode -asUtility condition`;
connectAttr -f ($distanceNode + ".distance") ($conditionNode + ···
        ".colorIfFalseR");
connectAttr -f ($distanceNode + ".distance") ($conditionNode + ···
        ".secondTerm");
```

A condition node is created and it's name is stored into the $distanceNode string. The 'distance' output from the distance node is then connected to the '.colorIfFalseR' and '.secondTerm' attributes on the condition node using the 'connectAttr' MEL command.

```
//Set the condition node operation to 'greater or equal' ie, (>=)
setAttr ($conditionNode + ".operation") 3;
//Set the condition node's first term equal to the $totalDistance
setAttr ($conditionNode + ".firstTerm") $totalDistance;
//Set the condition node's colorIfTrueR equal to the $totalDistance
setAttr ($conditionNode + ".colorIfTrueR") $totalDistance;
```

The rest of the condition node is setup using the 'setAttr' command. By setting the 'operation' to '3', we are setting it to be 'greater or equal'. The $totalDistance (which is the initial length of the chain) is plugged into the '.firstTerm' and the 'colorIfTrueR'.

From here, we need to take the output of the condition node, normalize it through a multiply/Divide node, then plug it directly into the scaleX of each joint in the IK chain. Start by creating a multiply/Divide node:

```
//Create a multiply/Divide render node.
string $muliDivNode = `shadingNode -asUtility multiplyDivide`;
```

This line creates a multiply/Divide utility node and stores it's name into the $multiDivNode string variable. By default, the node is set to perform a 'multiply' operation. To change it to 'divide', simply set the 'operation' variable to '2':

```
//Set the node operation to 'divide'
setAttr ($muliDivNode + ".operation") 2;
```

The dividend needs to be equal to the '.outColorR' attribute on the condition node. The divisor must equal the initial length of the chain ($totalDistance).

```
//Set the dividend to be the distance btw the ik handle/start joint.
connectAttr -f ($conditionNode + ".outColorR") ($muliDivNode + ".input1X");

//Set the divisor to the total distance along the chain
setAttr ($muliDivNode + ".input2X") $totalDistance;
```

With this setup, the multiply divide node will return a value of '1' when the condition node returns a length equal to the $totalDistance float. This is what we want. Unless the joint needs to stretch to reach the IK handle, the joint's scaleX attributes should be equal to '1' (not scaled). So now that we have the properly normalized scale factor, let's plug this into the scaleX of each joint in the chain. To do so, we will need to use another while loop to iterate over each joint in the chain. We are going to use the same exiting condition as the first loop. Let's get our loop variables setup and make the initial selection:

```
$exitLoop = 0;
$currentJoint = $startJoint[0];
select $currentJoint;
```

With the start joint selected and the $exitLoop variable set to 'o' we are ready to enter the new loop.

```
//The while loop keeps going until the current joint equals the end joint.
while ($exitLoop == 0)
{

        //Connect the output of the multiply/divide node to the
        //scale 'X' of the joints. This will cause them to stretch
        //along their length as the distance expands.
        connectAttr -f ($muliDivNode + ".outputX") ($currentJoint + ".scaleX");

        //Pickwalk down to move down through the joint heirarchy.
        pickWalk -d down;
        $sel = `ls -sl`;
        $currentJoint = $sel[0];

        //Exit loop condition
        if ($currentJoint == $endJoint[0])
        {
                $exitLoop = 1;
        }
}
```

The loop is exactly like the first one except the exiting condition has been moved to the end of the loop (after all the operations). This is to prevent the last joint from getting it's scale attribute connected. This is often the desired behavior.

As the loop iterates, it connects the multiply/Divide node's '.outputX' into the '.scaleX' of each joint. After making the connection, the loop pick walks downwards (to select the next joint) and continues on.

The script is now finished. To make it a little bit more user friendly, add one last line that selects the IK handle for the user. The user will likely want to test the behavior of the chain and selecting it for him/her is a nice little touch.

```
select $ikHandle;
```

■ Finished Script

This exercise demonstrated the creation of a script with NO error checking whatsoever. As was already discussed, this was purposely done because error-handling can severely reduce the readability of your code (and make it harder to teach). The version of the script on the DVD includes full error handling. As good practice, try adding some error handling to the script to prevent the user from making erroneous selections before attempting to execute the 'Make Stretchy' button.

Final Thoughts:

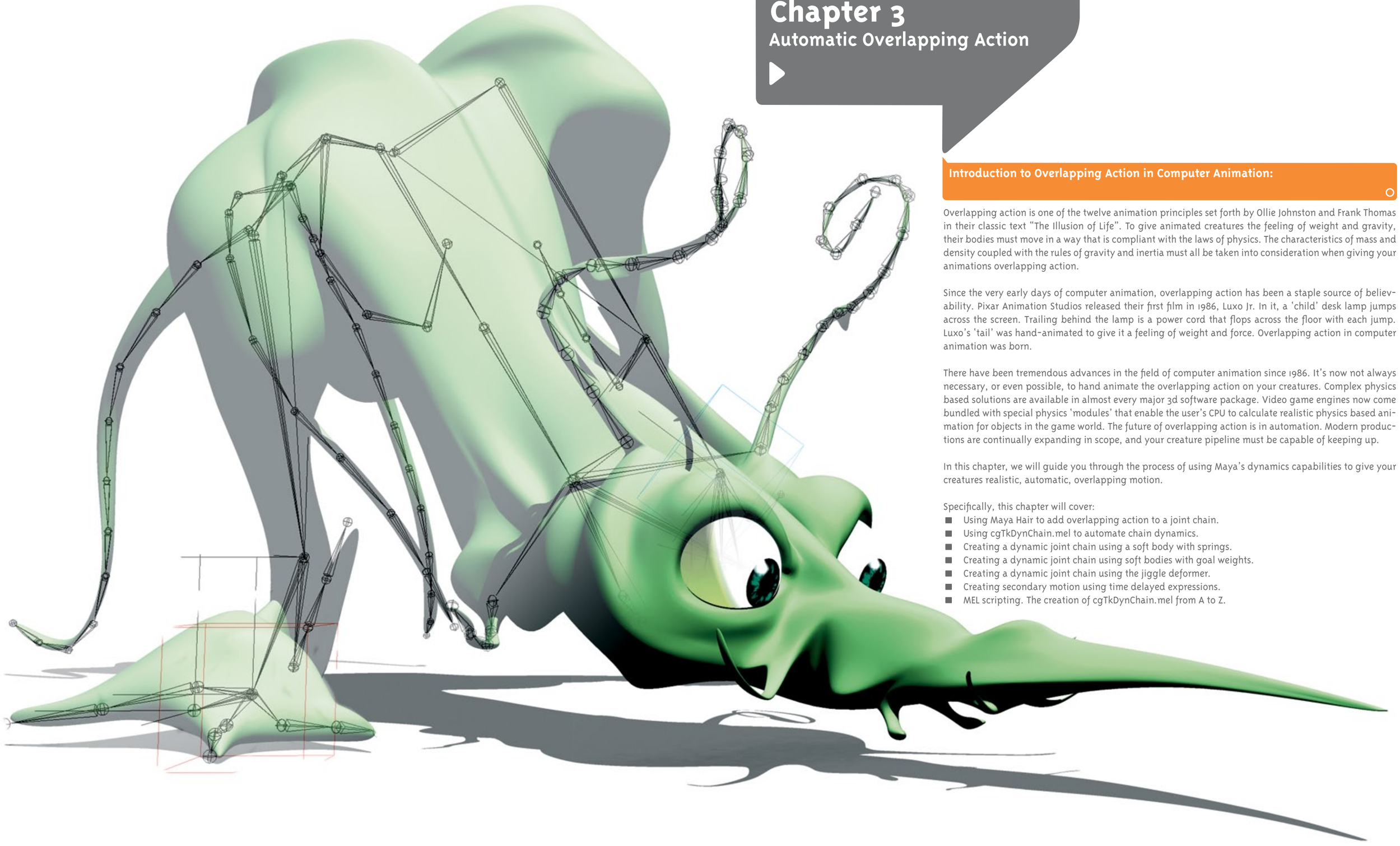
By the end of this chapter, you should have a solid understanding of what squash and stretch can do for your animations. In addition to learning how to make your digital creatures squash and stretch, you have also been shown how to automate the process by using MEL.

Squash and stretch is such a huge topic, it would be impossible to cover every definition of the concept in one chapter. That being said, there are some fundamental tools that can be used to get a soft, organic quality in your creatures. Stretch joints will allow your creatures to be posed beyond their apparent physical limitations. Combined with cleverly constructed blendshapes, your creatures can take on a truly 'cartoony' appearance.

Squash and stretch principles will be revisited again in chapter four and six where you will see how to successfully apply these ideas to the face and body. Before moving on from this chapter, please remember that almost all of these ideas can be applied in some form or another to realistic (non-cartoon) characters as well. The spine setup from this chapter can really bring an organic quality to even the most hyper-real creatures.

This chapter introduced you to one of the fundamental principles of animation, Squash and Stretch. The next chapter will introduce another principle, that of overlapping action. Like every chapter in this book, the next one will demonstate several different methods of enhancing your digital creatures while providing a quick and easy way to apply the techniques in a production environment.

Chapter 3
Automatic Overlapping Action



Introduction to Overlapping Action in Computer Animation:

Overlapping action is one of the twelve animation principles set forth by Ollie Johnston and Frank Thomas in their classic text "The Illusion of Life". To give animated creatures the feeling of weight and gravity, their bodies must move in a way that is compliant with the laws of physics. The characteristics of mass and density coupled with the rules of gravity and inertia must all be taken into consideration when giving your animations overlapping action.

Since the very early days of computer animation, overlapping action has been a staple source of believability. Pixar Animation Studios released their first film in 1986, Luxo Jr. In it, a 'child' desk lamp jumps across the screen. Trailing behind the lamp is a power cord that flops across the floor with each jump. Luxo's 'tail' was hand-animated to give it a feeling of weight and force. Overlapping action in computer animation was born.

There have been tremendous advances in the field of computer animation since 1986. It's now not always necessary, or even possible, to hand animate the overlapping action on your creatures. Complex physics based solutions are available in almost every major 3d software package. Video game engines now come bundled with special physics 'modules' that enable the user's CPU to calculate realistic physics based animation for objects in the game world. The future of overlapping action is in automation. Modern productions are continually expanding in scope, and your creature pipeline must be capable of keeping up.

In this chapter, we will guide you through the process of using Maya's dynamics capabilities to give your creatures realistic, automatic, overlapping motion.

Specifically, this chapter will cover:

- Using Maya Hair to add overlapping action to a joint chain.
- Using cgTkDynChain.mel to automate chain dynamics.
- Creating a dynamic joint chain using a soft body with springs.
- Creating a dynamic joint chain using soft bodies with goal weights.
- Creating a dynamic joint chain using the jiggle deformer.
- Creating secondary motion using time delayed expressions.
- MEL scripting. The creation of cgTkDynChain.mel from A to Z.

Using Maya Unlimited's Hair Dynamics for Overlapping Action:

Since version 6.0, Alias Maya has included Hair as a part of their Maya Unlimited package. User's without Maya Unlimited version 6.0 or greater will not be able to take advantage of this feature. It is the authors opinion that Hair is the most significant addition to Maya in a long time and, in most productions, it can be used enough to warrant an upgrade purchase if needed.

Maya Hair is deceptively named to lead one to believe that it's sole purpose is to animate hair. Perhaps a better name would be 'Maya Curve Dynamics' since what it effectively does is give soft-body properties to Maya CV curves. These curves then drive paint effects to create hair.

So the problem then becomes, how can I get the dynamic motion of a hair curve to affect my creatures? Luckily there is a way to do this by using a spline inverse kinematics solver that is attached to a series of joints. For those not familiar with spline IK , it is an inverse kinematics solver that uses a curve to control the orientation of the joints. So using Hair dynamics on a joint chain is as simple as making the spline ik curve a dynamic 'hair curve'.

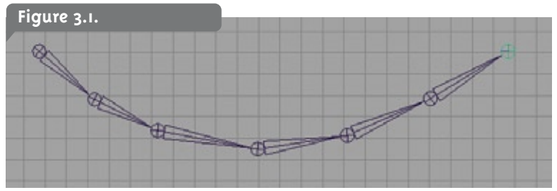
The process is straight forward and simple but also quite time consuming and tedious. As is the case with most tedious processes, this technique lends itself well to being automated by a MEL script. Exercise 3.1 will demonstrate how to setup a joint chain with Maya Hair, manually. Then, in exercise 3.2, you will be guided through the use of a mel script that will automate the process.

Exercise 3.1 — Setting-Up a Dynamic Joint Chain with Maya Hair (manually)

In a tight production schedule, animators can use any extra time to tweak performances and poses. If that extra time must be spent animating floppy appendages like tails and antennae, the performance may suffer. A dynamic chain of joints can be useful in a wide range of situations to help give your animations that extra bit of realism.

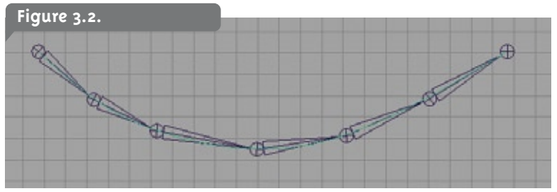
1. In a new Maya scene file, drop into an orthographic viewport and draw a chain of four to ten joints in the shape of a tail. FIGURE 3.1.

Figure 3.1.



2. We now need to draw a curve through the joint chain to use as a hair. Use the menu item, `Create > CV Curve` and hold down the 'V' key to snap a control vertex to each joint. At the last joint, press `Enter` to end the curve. You should now have a curve that is precisely drawn through the joint chain. FIGURE 3.2.

Figure 3.2.



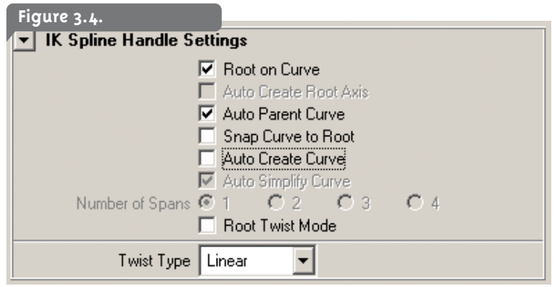
3. With the curve selected, execute the menu command `Hair > Make Selected Curves Dynamic`. The relevant hair and follicle nodes are created and your hypergraph should now look like figure 3.2. FIGURE 3.3.

Figure 3.3.

4. If you hit 'play' now, you can observe the curve that is colored blue, stretch and settle, as though it is nailed at each end. This is the curve that Maya Hair is now affecting, and we need to attach it to the joint chain.

5. To attach the dynamic curve to your joint chain, we are going to use an IK spline solver. Under the animation menu set, select `Skeleton > Ik Spline Handle Tool`. By default, an IK spline solver will create and use it's own curve. To disable this feature, uncheck the 'Auto Create Curve' option box in the IK spline tool options. FIGURE 3.4.

Figure 3.4.

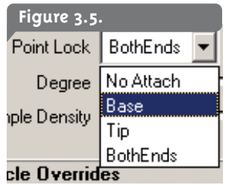


6. Now activate the IK spline tool and click on the base joint, the tip joint and the dynamic curve, in that order. This will create an IK spline handle at the tip of your joint chain. The IK solver will now cause the dynamic curve to control the orientation of the joints.

7. The chain will currently hang as though each end is fixed. This kind of behavior is useful in some instances (like a swinging bridge effect). In order to have it hang from the base joint, you must edit the follicle node. Select the dynamic curve (colored blue) and in the attribute editor find the follicle shape node. Under the follicle shape node, find the 'point lock' attribute and set it to 'base'. Now the chain will hang from the base joint.

8. In order to adjust the physical properties of your chain, you must go back under the follicle node and check 'Override Dynamics`. This will cause the follicle node attributes to control the behavior of the chain. FIGURE 3.5.

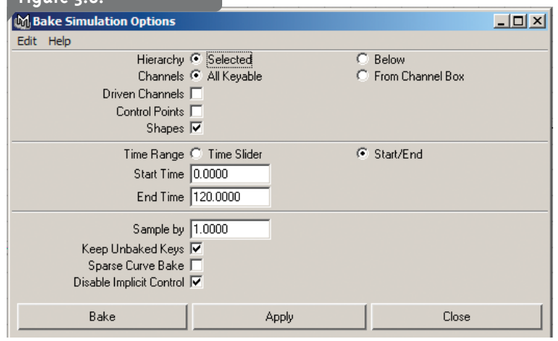
Figure 3.5.



9. Setting the stiffness to a very low value (0.001) will result in chain behavior like that of a thick rope. Higher stiffness values can be used to simulate appendages like antennae.

10. In order to use this animation for video games, you must be able to bake out the animation into keyframes on the joint's rotation channels. To this author's knowledge, there are no game engines that currently support Hair dynamics. To bake, select all your joints in the hypergraph and execute `Edit > Keys > Bake Simulation` . In the options box, specify a frame range and then hit 'bake' FIGURE 3.6 . A keyframe will be set on every frame effectively 'recording' the dynamics. You are now free to delete the hair nodes and the IK handle without destroying the animation.

Figure 3.6.



There are other very interesting effects that can be achieved by utilizing the various attributes found on both the follicle node and the hair system node. You can create effects to simulate ropes and chains by setting the stiffness and drag to very low values. Similarly, you can make your chain behave like a springy piece of wire by increasing the stiffness, and lowering the damping. If your character has highly stylized hair strands, you could rig them with joints and get them flopping around in the breeze by adjusting the turbulence factor in the hair system node. Don not forget that hairs can also collide with any surface. A myriad of effects can be achieved by adjusting the following attributes:

Stiffness: The most obvious attribute, stiffness simply controls how much the curve will resist bending. It has a range from 0 to 1 with 1 being completely stiff and 0 being completely loose.

Damping: Damping is a common attribute found in every dynamic simulation. With Hair, damping is no different. It will cause the hair's kinetic energy to dissipate with time. Damping values range from 0 to 100. If your hair curve is behaving in an erratic, chaotic manner, try upping this value by increments of 10-20. Different materials will 'damp' more than others, you have to keep this in mind as you adjust a dynamic simulation.

Drag: It is best to think of this attribute as controlling the viscosity of the medium that the hair is flowing through. Low values (0.05) can simulate the drag from air. Higher values can make it look as though your chain is flowing through water or even molasses. Drag values range between 0 and 1.

Friction: This attribute only comes into effect when your hair is colliding with other surfaces. The friction attribute ranges between 0 and 1 with 0 being no friction at all. Be forewarned that high friction values can cause the hair to 'stick' to surfaces and behave in a very erratic fashion. Use with caution.

Gravity: Values range from 0 to 10. A value of 1 usually works well to simulate real world gravity.

Turbulence: This attribute can be used to add a little bit of random motion to your chain. You can use it to get gusting wind effects, or subtle oscillations like those of a lite breeze. Adjust the strength value between 0 and 1 to control the amount of turbulence. The frequency and speed values can give you added control over the chaotic forces on the chain.

Automating Setups with MEL:

Exercise 3.1 demonstrated a fairly simple setup that could be used in many situations throughout a production. As a general rule, whenever you have to do something more that twice, it is best to script it.

Even if you are a beginner to MEL scripting or even programming, it is still worth the time investment to start automating your setups, regardless of the long time it will take you at first. Consider writing scripts as an investment in your pipeline. If after each production you find yourself with three or four new tools, you will soon find subsequent productions going much smoother as your workflow steadily evolves.

Now that you have seen the process used to add hair dynamics to a joint chain, you can appreciate the advantage of having that process automated. Included on the DVD, is a MEL script that was written to do just that. You can find it under:

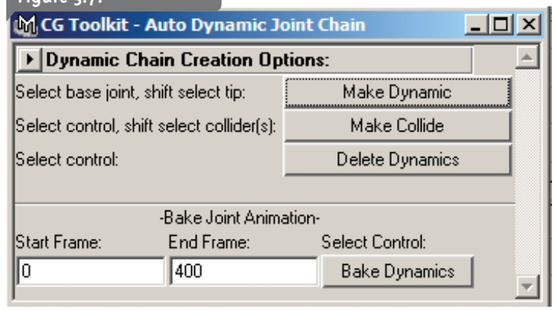
MEL Script Files/cgTkDynChain.mel cgTkDynChain.mel

To use the cgTkDynChain.mel MEL script, copy it into your scripts directory, (user/My Documents/maya/scripts, in Windows) restart Maya and execute the following from the command line:

cgTkDynChain;

The 'Automatic Dynamic Joint Chain' window will pop up FIGURE 3.7. This window contains all of the functions necessary to add hair dynamics to a chain of joints.

Figure 3.7.



Because of the power that scripting affords the setup artist, we have chosen to emphasize MEL usage as part of the training in this book. To help you in the creation of your own scripts, we have included a very thorough breakdown of the cgTkDynChain.mel script at the end of this chapter.

Exercise 3.2

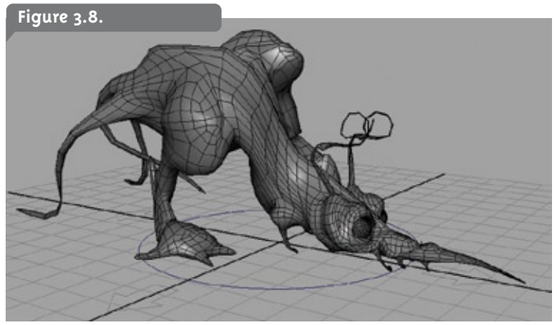
Using CG Toolkit's Automatic Dynamic Joint Chain Script

This exercise will introduce you to cgTkDynChain.mel. You will use the script to add automatic overlapping action to the floppy appendages of a bizarre creature named Johnny Poopooha. To watch the instructor completing this exercise, please watch the exercise 3.2 video included on the DVD.

1. If you haven't done exercise 3.1, we recommend doing that first so that you understand what is going on in the background as the script executes.

2. Start by opening the exercise3.2_Start.mb file. This file contains the Johnny Poopooha character FIGURE 3.8 . If you playback the animation, you will see his walk cycle over 24 frames. The animator has finished his work and now needs the automatic overlapping action added to the mix. For a creature TD, this is a common scenario during a production.

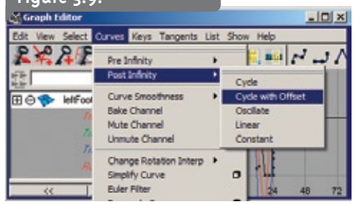
Figure 3.8.



3. We are going to use the cgTkDynChain.mel script to quickly setup Johnny's tails and antennae to flop around dynamically. This script uses the same technique outlined in exercise 3.1, and a complete line-by-line breakdown of the script is included in the last section of this chapter. You can find cgTkDynChain.mel in the MEL script's folder. Please copy it into your scripts directory and restart Maya. To use it, simply type 'cgTkDynChain;' into the command line and hit 'Enter'.

4. Before running the script, we need to cycle the animation of Johnny's walk so that it extends beyond 24 frames. This way, we can let the dynamics settle into a rhythmic pattern that will loop properly. To do this we are going to use the 'Cycle with offset' command, in the graph editor, `Window > Animation Editors > Graph Editor`. We need to cycle all of the animation on Johnny's controllers. To do so, select each controller, then in the graph editor, drag select over the channels in the left column. With all the curves selected, go under the graph editor's menu to `Curves > Post Infinity > Cycle with Offset` FIGURE 3.9. This will cycle the animation on the selected curves. Please do this for the following controller objects, being sure to cycle *all* of the channels for each controller.
leftFootControl
rightFootControl
COG_Control
leftKneeControl
rightKneeControl

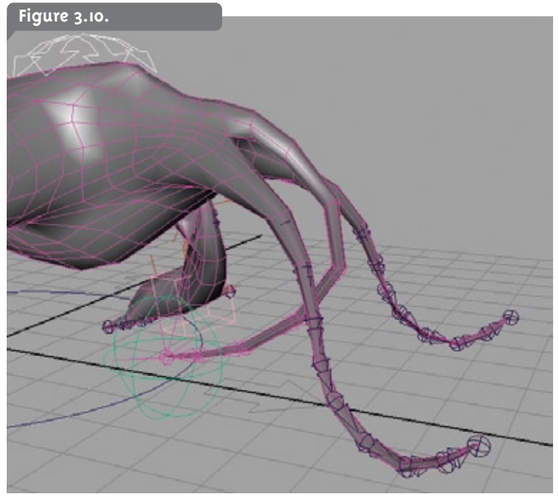
Figure 3.9.



5. Now extend the frame range to about 360 frames. This will give us an adequate range that will allow the dynamics to form into a rhythmical pattern. If you try to run the dynamic chains using only the 0 to 24 frame range, the dynamics will not have enough time to catch up and it will be impossible to get a proper looping motion on the appendages. Playback to ensure that the animation on all the controllers has been properly looped to infinity. If you notice any change in the walk cycle after frame 24, it is likely that an animation curve was not properly looped. Lastly, hide the controller curves by selected the circle around the base of the creature and setting the 'Control Visibility' attribute to 'off'. This will reduce the clutter in the scene. Open exercise3.2_MiddleA.mb to see the finished looping animation.

6. We are now ready to start setting up the tails. Run the script by executing 'cgTkDynChain;' from the command line. In order to make a chain of joints dynamic, you must first select the base joint, then the tip, and hit the 'Make Dynamic' button FIGURE 3.10 . A controller sphere will appear at the tip of the chain. This controller houses all the attributes needed to adjust the behavior of the chain.

Figure 3.10.

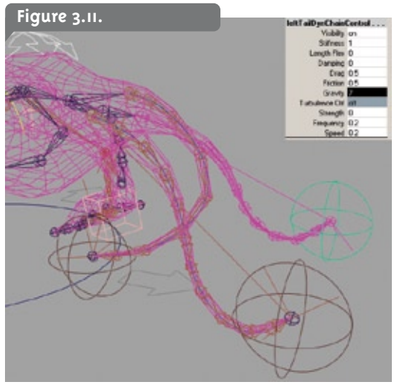


7. Start by setting up his middle tail. Select the joint near the base of his middle tail, named 'tailA'. Shift select the joint at the end of the middle tail, named 'tailTip'. Hit the 'Make Dynamic' button in the script window. Grab the controller sphere at the tip of the tail and set the following:
Stiffness = 0.3
Gravity = 5.0

8. Playback the animation to observe the tail as it flops around. Try adjusting the different attributes on the controller sphere and observing the affect they have on the behavior of the tail.

9. Now let's setup the left and right tails. Create a dynamic chain from the 'leftTailA' joint to the 'leftTailTip' joint. Create a dynamic chain on the corresponding joints for the right tail as well. If you playback right now, the tails will fall down and completely loose shape. We need to adjust the settings to get some nice behavior. Set the following on both tail controller spheres FIGURE 3.11.
Stiffness = 1
Drag = 0.5
Gravity = 7

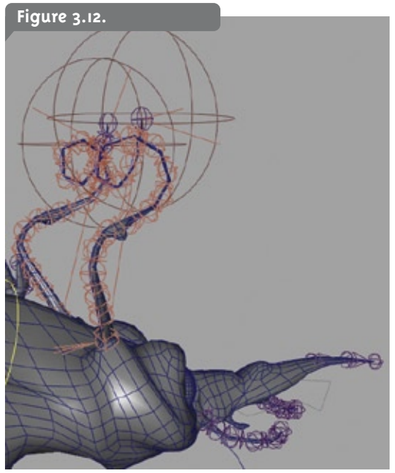
Figure 3.11.



10. The tails should now be flopping around quite nicely. Open exercise3.2_MiddleB.mb to see the animation thus far. To setup chains like these, you must get used to simply adjusting each setting and testing the motion until it looks the way you want it. Arriving at these specific settings required a little bit of trial and error.

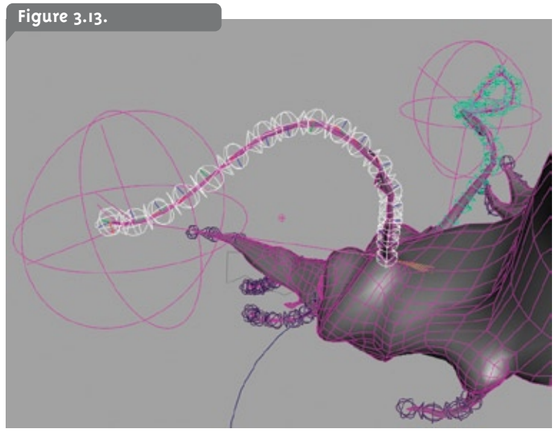
11. Moving on now, setup the antennae on Johnny Poopooha's head. Create dynamic chains from the bottom joints to the tips of each of the creatures antennae FIGURE 3.12. If you playback now, you will notice that they completely lose shape and fall through his head. Let's crank up the stiffness on both antennae controllers to a value of '1'.

Figure 3.12.



12. The antennae still flop around too chaotically FIGURE 3.13 . They need less force on them. Turn off the gravity by setting it to '0'. This has helped some, but the antennae are still quite out of control. In order to help these appendages flop around while maintaining some structure, we need to take advantage of the damping attribute. Set the damping on each antennae to '25'.

Figure 3.13.

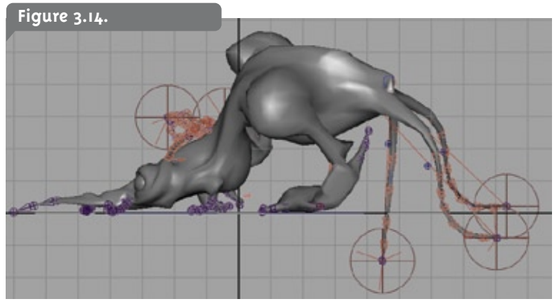


13. The antennae are now lightly bouncing with the motion of Mr. Poopooha's head. Open exercise3.2_MiddleC.mb to see the progress after step twelve.

14. There are several other little appendages on Johnny's head and neck that can be setup as well if you want more practice. Once you are happy with the motion of the chains, we need to move on to setting up some collisions.

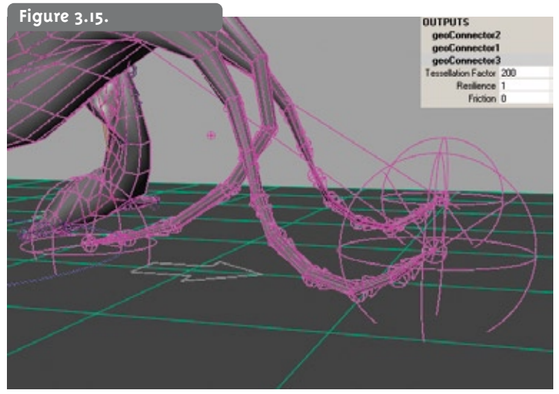
15. You may notice that his tails are falling through the ground plane (the grid)FIGURE 3.14 . To prevent this, let's use a polygon plane to act as the ground for the chains to collide with. Create a polygon plane and set the scale x,y and z to '25'. Select a controller sphere on his tail then shift select the ground plane and hit the 'Make Collide' button. This will attach the polygon plane as a collision object. Do this for all three tails.

Figure 3.14.



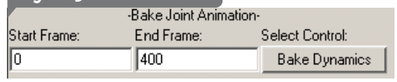
16. Drop into a side viewport to see his tails colliding with the ground. There are a couple parameters that can be adjusted to tweak the amount of friction between the tails and ground plane. Select the ground plane and click on one of the 'geoConnector' outputs FIGURE 3.15 to see the adjustable attributes. 'Resiliance' will control the amount of energy that is absorbed by the colliding chain. A value of '0' will cause no bounce whatsoever, a value of '1' will cause the hair to retain one hundred percent of it's energy. Values greater than '1' will add speed to the colliding hair. 'Friction' will control the velocity of the colliding hair, parallel to the surface. A value of '1' will cause the hair to bounce straight off the surface at a ninety degree angle. Resilience and friction work together to control how your chains will collide. For the purposes of Johnny Poopooha's walk cycle, the default values will work fine.

Figure 3.15.



17. We are now at the point where we are ready to bake out the animation into the joints. Open exercise3.2_MiddleD.mb to see Johnny's animation before baking. By 'baking' I mean setting a keyframe on every joint for every frame. This step is absolutely necessary if Johnny is to be imported into a game engine with his fancy new tail motion. Film and television pipelines will likely need to have the animation baked into the joints as well, unless every render node in the render farm is capable of calculating Hair dynamics. Regardless, it is always good practice to hand off a clean animation file and there is no reason to keep the dynamic chains in this scene any longer. To bake the animation on a dynamic chain, be sure to specify the frame range of 0 to 360, then simply select the controller sphere, and hit 'Bake Dynamics' FIGURE 3.16 . The script will run through the animation and record the dynamics into keyframes. Do this for every chain on Johnny's rig.

Figure 3.16.



18. Once all of the baking is finished, you need to clean up the hypergraph by removing any nodes that are associated with the hair dynamics. Fortunately, this can be done quite easily by using the 'Delete Dynamics' button in the script. Select a controller sphere on a dynamic chain, then hit 'Delete Dynamics'. All of the nodes associated with that chain will be removed, leaving only the joints. You can also delete any collider objects in the scene. Search through the hypergraph to ensure that all of the hair nodes have been successfully deleted.

19. Because we are trying to add the dynamic motion to a game cycle, we need to cut the animation back down to 0 to 24 frames. Do do this, you will need to find an interval of keyframes to copy and paste into the beginning of the animation. The dynamic motion becomes very rhythmic as the animation goes on. Copy a 24 frame chunk of keyframes from frame 288 to 312. This is the twelfth cycle (24x12=288) and by this time, the appendages are flopping in a near perfect rhythm. You may need to hand tweak the beginning and end of the cycle to help them line up perfectly. Johnny Poopooha's overlapping action is now finished. Open exercise1.2_Finished.mb to see the final animation. Notice that because the animation is all baked, you can scrub in the timeline without the appendages flipping out.

Dynamic Overlapping Action Without Hair:

While using Maya Hair is certainly one of the best methods of adding dynamic motion to your creatures, it is not the only way. Maya has several different dynamics options, most of which are included in the Maya Complete edition, unlike Hair which is only in Maya Unlimited.

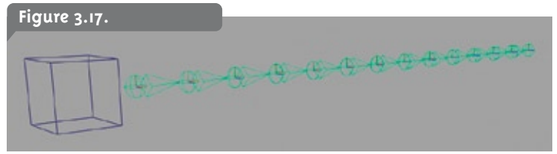
The following exercises demonstrate other techniques for creating dynamic joint chains, some of which are better, in certain situations, than Hair. After completing these exercises, you will be familiar with many different ways of giving your creatures believable dynamic motion. It's always important to have a very broad tool-set to assist in tackling any problem associated with character rigging.

Exercise 3.3 Creating a Dynamic Joint Chain Using Soft Bodies and Springs

This exercise will use a spline ik solver to control a chain of joints. The spline ik curve is then made a softbody. This softbody is then rigged up with springs to help the chain hold it's shape.

1. Open excercise3.3_Start.mb. This scene file contains a little joint chain that is going to be rigged with a softbody spline IK FIGURE 3.17.

Figure 3.17.

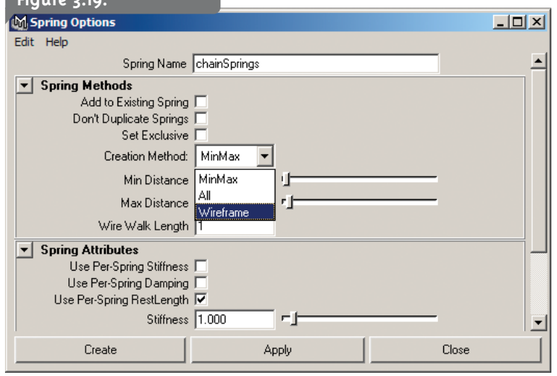


2. From the animation menu set, select `Skeleton > Ik Spline Handle Tool` . Reset the tool options and then click once on the first joint in the chain and click a second time on the last joint. The spline IK handle will be created.
3. Now look in the hypergraph and you should see a curve has been created along with the IK handle. Name this curve 'dynamicCurve'.
4. Select 'dynamicCurve' and under the dynamics menu set, select `Soft/Rigid Bodies > Create Soft Body` . In the create softbody options window, reset the tool options and click 'Create'.
5. You will notice that your dynamic curve now has a particle node underneath it FIGURE 3.18 . This particle node is our access to all of the dynamic capabilities in Maya. If you look closely, you will notice that each vertex on the dynamic curve has a particle on it (visible as a small gray square in the viewport). The control vertices on our dynamic curve are effectively parented to these particles so that wherever the particles go, the corresponding CV will follow .

Figure 3.18.

6. To get these particles to move about and behave in a 'floppy' manner, we must add springs to them. A 'spring' is simply a relationship between two particles that causes the attached particles to react to one another. Using springs, we can help our chain maintain shape while undergoing forces. The springs will also be necessary for attaching the chain to a parent hierarchy. To add springs to your dynamic curve particles, simply select the curve and execute `Soft/Rigid Bodies > Create Springs`. In the create springs option box, reset everything and change the creation method from 'MinMax' to 'Wireframe'. Set the spring name to 'chainSprings' and then hit 'Create' FIGURE 3.19.

Figure 3.19.

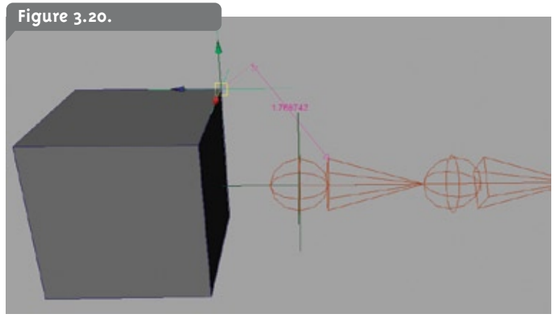


7. Now all of the particles in the chain have springs between them. These will hold the particles in the chain together. In order to attach the whole chain to an external hierarchy, we must attach the particle at the base joint to the vertices on our polygon cube. This cube can then be parented into your rig hierarchy to enable the chain to inherit parental motion. To do this, we are going to use the create springs tool with the 'MinMax' creation method as opposed to the 'Wireframe' method that we used

on the chain links.

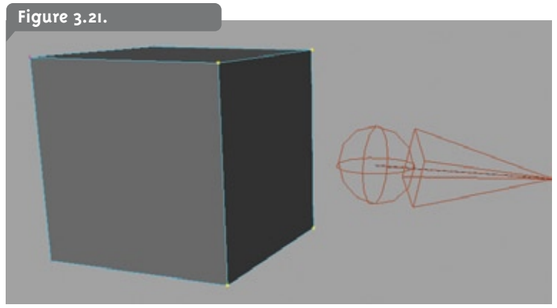
8. The 'MinMax' creation method will create springs that fall within the allowed range as specified in the 'Create Springs' option box. To determine the min and max distance settings, you can use a 'Distance Tool'. Select `Create > Measure Tools > Distance Tool`. Hold down the 'V' key and click once on the root joint and once on a top corner of the polygon cube. This will result in the creation of a distance tool . For our scene file, the distance is about 1.77 units from the first particle to the corner of the polygon cube that we want to attach it to FIGURE 3.20.

Figure 3.20.



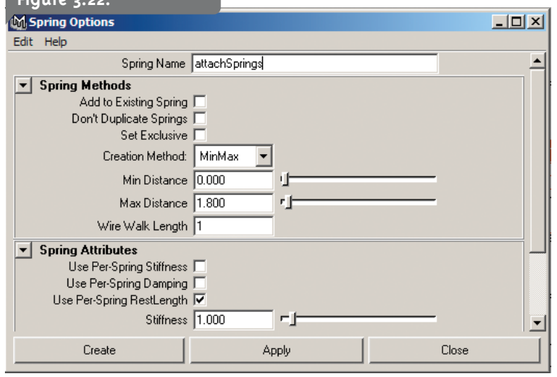
9.Now we are ready to add more springs. Set your selection masks so that you can select the particle at the base of the joint chain. With this particle selected, shift select all four corner vertices on the polygon cube FIGURE 3.21.

Figure 3.21.



10. Select `Soft/Rigid Bodies > Create Springs`. In the create springs options box, set the creation method to 'MinMax' and enter a value of 1.8 into the 'Max Distance' attribute. This will ensure that only springs between the last particle and the cube are created. If the max distance is not set appropriately, your chain may have stray springs that make it difficult to tweak the behavior. Set the 'Spring Name' to 'attachSprings' and hit 'Create' FIGURE 3.22.

Figure 3.22.



11. Select your particle node in the hypergraph and select `Fields > Gravity`.
12. Select your particle node again and select `Fields > Drag` . Step eleven and twelve will result in the creation of two dynamic fields that will affect the particles FIGURE 3.23.

Figure 3.23.

13. Hit play to observe the behavior of the dynamic chain. To see how your chain should now behave, open 'exercise3.3_Middle.mb' and hit play. It will fall (because of the gravity) and flop around, but the chain springs are far too loose.
14. Grab the 'chainSprings' node in the hypergraph and enter a value of '100' into the stiffness attribute in the channel box FIGURE 3.24. The chain will now behave a little better but the attach springs are still far too loose.

Figure 3.24.

SHAPES	
chainSpringsShape	
End1 Weight	1
End2 Weight	1
Use Stiffness PS	off
Use Damping PS	off
Use Rest Length PS	on
Stiffness	100
Damping	0.2
Rest Length	0
Manage Particle Death	on

15. Grab the 'attachSprings' node in the hypergraph and enter a value of '100' into the stiffness attribute in the channel box just like with the chain springs.
16. The chain will behave much better now. However, you may notice that it swings about too much without ever dissipating energy (like a pendulum in a vacuum). This is where the 'Drag' field comes in handy. Select the drag field and set the 'Magnitude' to a value of about '10' FIGURE 3.25.

Figure 3.25.

17. To attach this chain to a rig, parent the polygon cube into your rig hierarchy. For example, if this were a pony tail, parent the polygon cube under the head joint on the character.
18. You may wish to have this chain collide with other surfaces. To do so, select the particle node and shift select a collision surface then execute, `Particles > Make Collide`. You can specify a friction attribute to control collision behavior.
19. To see the finished rig in motion, open exercise3.3_Finished.mb and playback.

A Couple Tips:

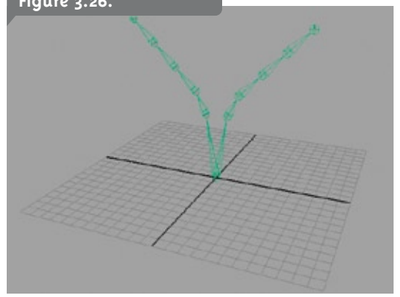
- You can tweak several values to achieve different effects. Try adjusting the spring stiffness first. The 'conserve' attribute on the particle node can help settle down spring simulations that are jittery or too fast. Try setting it to values between 0.9 and 1.0.
- The 'Dynamics Weight' attribute on the particle node can help too, try experimenting with it.
- You may wish to have more CVs on your chain than what the IK spline tool creates. You can create you own curve and use that by disabling the 'Auto Create Curve' option in the IK spline tool options box.
- Because collisions take place between the particles and your collision objects, you may need to use dummy collision objects depending on the thickness of the mesh that is bound to your joints. This can prevent unwanted 'crashing'.

Exercise 3.4 — Using Soft Bodies with Goal Weights for a Dynamic Joint Chain

This is a far simpler method of creating a dynamic joint chain than that of exercise 3.3. It is, however, somewhat limited in it's behavior. Goal weights result is much more linear motion than that of springs. While the spring method might be better suited for a tail, the goal weight method in this tutorial is great for appendages like antennae.

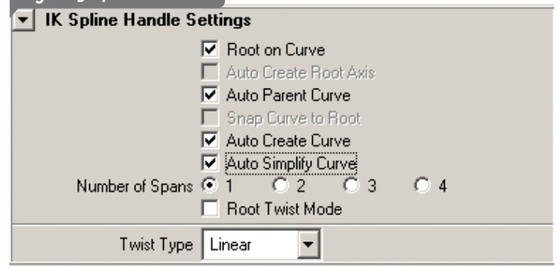
1. Open exercise3.4_Start.mb. This file contains a series of joints in the shape of two antennae. A sin curve was used to give the head joint some oscillating motion, like you might get when a character is walking. The sin expression was then baked out into the head joint in the form of keyframes. Playback the scene to see the results FIGURE 3.26.

Figure 3.26.



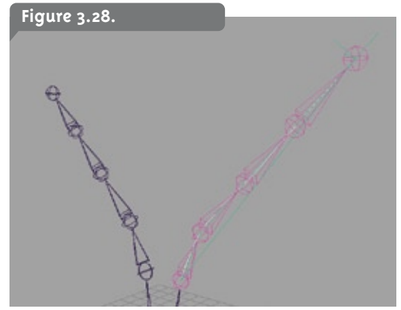
2. We need to setup the antennae such that they bounce around with the head joint. Select `Skeleton > Ik Spline Handle Tool`. In the tool options, ensure that 'Auto Simplify Curve' is checked on. This will ensure that our curve has enough resolution to flop around, without having extra CV's that clutter up the dynamics. FIGURE 3.27.

Figure 3.27.



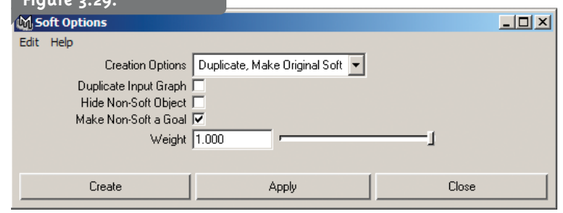
3. Now add an IK spline solver to an antenna by clicking on the start and end joints. An IK spline handle will appear at the tip of the joint chain and you should notice a curve has been created that runs through the joints FIGURE 3.28.

Figure 3.28.



4. In the hypergraph, select the curve (most likely named 'curve1'). Under the dynamics menu set, select `Soft/Rigid Bodies > Create Soft Body`. In the soft options window, reset the settings and then change 'Creation Options' to 'Duplicate, Make Original Soft'. Check the option box labeled 'Make Non-Soft a Goal' and set the weight slider to a value of '1'. Then hit 'Create' FIGURE 3.29.

Figure 3.29.



5. This will result in the creation of a duplicate curve object that will automatically get parented under the head joint. It is this curve that will act as a magnet for the spline IK curve. So as the goal curve flops around with the motion of the head joint, the spline IK curve will attempt to keep up. The initial goal weight of '1' will cause the chain to follow it perfectly and no secondary action will be visible if you playback at this time. In order to get the antenna to flop around, you must edit the goal weights per vertex.

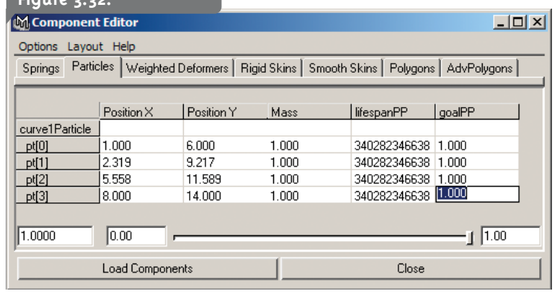
Figure 3.30.

6. We are going to select all of the particles along the length of the chain. To do so, hit the 'Select By Component Type' button FIGURE 3.30. Now click on the 'points' icon to uncheck it. Right-click on the 'Points' icon and check 'Particles' FIGURE 3.31 . This will enable you to select the individual particles. You can now safely marquee select over the entire joint chain. Ensure that all of the particles are selected.

Figure 3.31.

7. Select `Window > General Editors > Component Editor`. You will now see a spreadsheet view of the particle attributes. If necessary, click on the 'Particles' tab. Scroll over to the 'goalPP' column and notice that all of the values are currently set to '1' FIGURE 3.32.

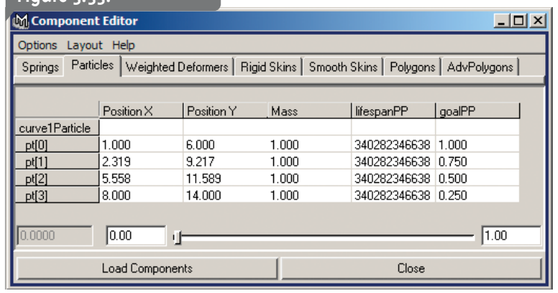
Figure 3.32.


Component Editor
Options Layout Help
Springs Particles Weighted Deformers Rigid Skins Smooth Skins Polygons AdvPolygons
Position X Position Y Mass lifespanPP goalPP
curve1Particle
pt[0] 1.000 6.000 1.000 340282346638 1.000
pt[1] 2.319 9.217 1.000 340282346638 1.000
pt[2] 5.558 11.589 1.000 340282346638 1.000
pt[3] 8.000 14.000 1.000 340282346638 1.000
1.0000 0.00 1.00
Load Components
Close

8. If you created the spline IK chain exactly as described in step 2, you should have a list of seven particles. The particle labeled pt[0] is the particle lying at the base of the joint chain, similarly pt[6] is the particle at the tip of the chain. Knowing this, we can adjust the weights to give the chain a nice falloff effect. Set the following weights FIGURE 3.33:

pt[0] = 1.00
pt[1] = 0.85
pt[2] = 0.70
pt[3] = 0.55
pt[4] = 0.40
pt[5] = 0.25
pt[6] = 0.10

Figure 3.33.

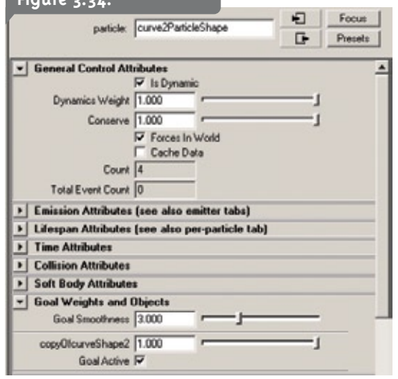

Component Editor
Options Layout Help
Springs Particles Weighted Deformers Rigid Skins Smooth Skins Polygons AdvPolygons
Position X Position Y Mass lifespanPP goalPP
curve1Particle
pt[0] 1.000 6.000 1.000 340282346638 1.000
pt[1] 2.319 9.217 1.000 340282346638 0.750
pt[2] 5.558 11.589 1.000 340282346638 0.500
pt[3] 8.000 14.000 1.000 340282346638 0.250
0.0000 0.00 1.00
Load Components
Close

9. Playback the animation to see the chain flop around. It should now be apparent exactly what affect the weight values are having on the particle's behavior. The antenna's joints should be flopping around until about frame 120 where the animation stops. When the joints stop flopping around, you can observe the dynamics settle out.

10. Take a look at excercise3.4_Finished.mb to see both of the antennae flopping around.

11. To get other effects, try adding various fields like drag or turbulence to the particle node. You may also find that adjusting the 'goal smoothness' and 'dynamics weight' attributes on the particle shape node can yield some interesting motion FIGURE 3.34.

Figure 3.34.



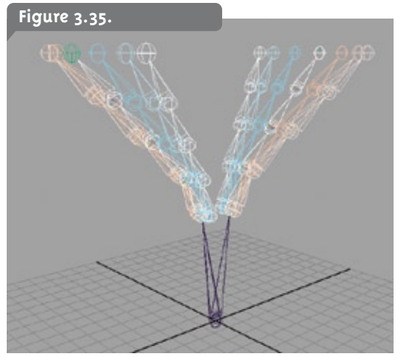
Up until now, we have only been using Maya's Hair and particle dynamics to achieve secondary motion. The last two exercises in this chapter use a deformer and an expression, respectively. Our hope is that by the end of these exercises, we will have armed you with enough tools to tackle any rigging problems involving automatic overlapping action.

Exercise 3.5 Using a Jiggle Deformer to create a Dynamic Joint Chain

This method yields almost the exact same results as exercise 3.4. The jiggle deformer can be a little bit more tricky to setup, and you may find that good results are harder to achieve. The main advantage with using a deformer is that you can scrub in the timeline to see the effects. You may wish to use this setup to give your animators something to see while they are animating. This is the jiggle deformer's only real advantage over soft bodies or Hair.

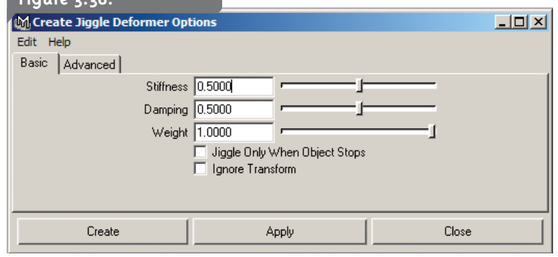
1. Open exercise3.5_Start.mb. This scene file contains a head joint with antennae. If you playback you will see the animation on the head joint as it bobs around until frame 45 where it stops. This setup is good for testing the behavior of your dynamic chain FIGURE 3.35.

Figure 3.35.



2. Like all the previous methods, this too utilizes an IK spline handle. Create one, with default options, from the base to the tip of an antenna.
3. Select the spline IK curve and from the animation menu set select `Deform > Create Jiggle Deformer`. Reset the options box and hit 'Create' FIGURE 3.36.

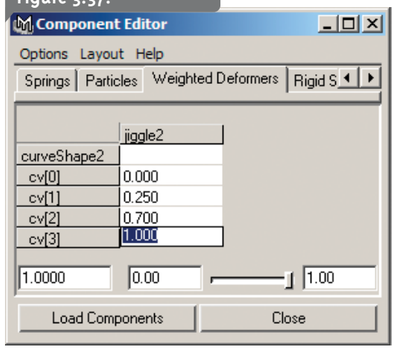
Figure 3.36.



4. Now select all of the control vertices along the length of your spline IK curve. With the vertices selected, go to `Window > General Editors > Component Editor`.
5. In the component editor, select the 'Weighted Deformers' tab. You should see four CV's that have values of '1' under the 'jiggle1' column FIGURE 3.37. Set these values to falloff towards the ends of the antenna.

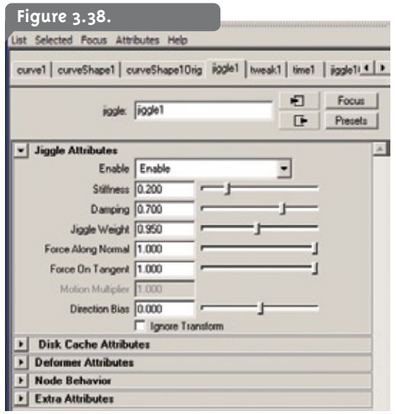
CV[0] = 0.00
CV[1] = 0.25
CV[2] = 0.7
CV[3] = 1.00

Figure 3.37.



6. If you playback now, you will see that the antenna is flopping around, but it's behavior is somewhat erratic and too chaotic. To settle the chain down, select the spline IK curve and find the 'jiggle1' node in the attribute editor. Set the following FIGURE 3.38 :
Stiffness = 0.2
Damping = 0.7
Jiggle Weight = 0.95

Figure 3.38.



7. Playing the animation should now result in a much better effect. Open the file, exercise3.5_Finished.mb, and hit play to see both antennae setup with this method.
8. Try experimenting with stiffness, damping and weight values to achieve different effects. Do not despair if your simulation goes completely wonky during the testing phase. The jiggle deformer is very particular and prone to wild behavior. That being said, a careful, patient, artist should have no problem taming it.

From here on, this chapter is going to make a minor detour and shift focus onto MEL based solutions. Beginners to programming may find some of the concepts in these lessons somewhat foreign. If you have done any amount of programming in the past, you should have no problem following along. If you have some experience with MEL, you should have no trouble whatsoever.

Most important of all, do not get discouraged. MEL can be daunting at first, but if you give it a proper time investment, it can give you the power to extend Maya in ways that are simply not possible through any other means.

As a final note, before I dive into the nitty gritty details of using MEL, I recommend that those users who do not feel comfortable reading or writing basic code, go out and pick up a programming specific book. There are several excellent MEL scripting specific texts, any of which can help prepare you for the MEL lessons in this chapter and later in this book. Of course, novice MEL programmers can still follow the exercise, as I will try to explain any potentially confusing concepts as I go along. Novices should check out the introductory MEL video on the DVD for a quick overview of MEL fundamentals that will help bring them up to speed. The MEL section in chapter one will also help those who are familiar will another language like C and who want to start using MEL.

Exercise 3.6 Using a MEL Expression to create a Dynamic Joint Chain

Expressions are MEL-based instructions, written to control an attribute over time. They are infinitely useful, and can be as simple or as complex as you like. You can write expressions that use logic to perform actions on your scene's objects. I've even seen expressions that create basic artificial intelligence behaviors.

Harnessing the power of expressions is essential for every creature setup artist. This exercise uses a time delayed expression on a series of joints, such that the animation on the first joint will drive the rest in a way that simulates a flapping effect. By the end of this exercise, you should have an appreciation for the power that expressions can give you.

1. Open exercise3.6_Start.mb. This scene file contains a joint chain that we want to use to simulate the motion of seaweed blowing in an underwater current. To start off with, we are going to use an expression to control the rotation of the root joint. Joints that are above the root, will be controlled by a separate expression FIGURE 3.39.

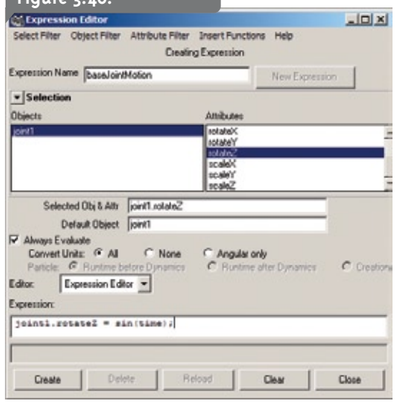
Figure 3.39.

2. Select the root joint (joint1). In the channel box, right click on 'Rotate Z', and choose 'Expressions'. This will bring up the expression editor. Name the expression 'baseJointMotion'. Then enter the following code into the expression editor text box:

```
joint1.rotateZ = sin(time);
```

Now hit 'Create' to create the expression FIGURE 3.40 .

Figure 3.40.

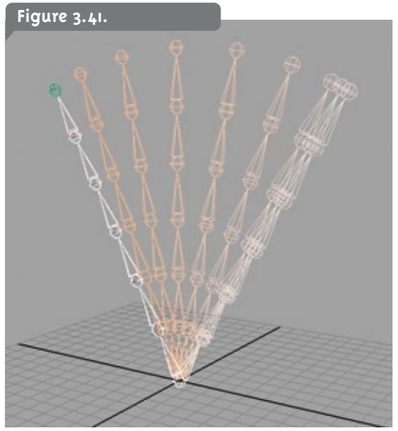


3. If you now hit play, the root joint will rotate back and forth between values of '1' and '-1' in the z axis. This is the nature of sinusoidal expressions. To adjust the range (and make the joint rotate more), we can multiply the expression by a factor. Let's try a factor of 30. Change the expression to the following and hit 'Edit'.

```
joint1.rotateZ = sin(time)*30;
```

4.You will notice now that as you playback, the root joint will rotate between '30' and '-30' degrees in the Z axis. This motion will work fine for now FIGURE 3.41. Open exercise3.6_Middle.mb to see the setup this far.

Figure 3.41.

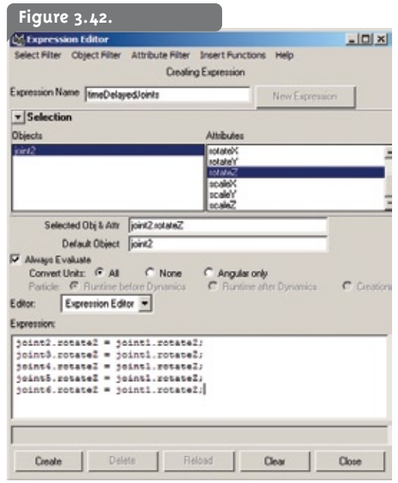


5. To get the rest of the joints 'flowing' with the motion of the root joint, we are going to use what is referred to as a 'time delayed' expression. Start by selecting the second joint in our seaweed chain and right-click on the 'Rotate Z' attribute, then choose 'Expressions'. In the expression editor, name this expression 'timeDelayedJoints'. Enter the following into the expression field:

```
joint2.rotateZ = joint1.rotateZ;
joint3.rotateZ = joint1.rotateZ;
joint4.rotateZ = joint1.rotateZ;
joint5.rotateZ = joint1.rotateZ;
joint6.rotateZ = joint1.rotateZ;
```

Hit the 'Create' button FIGURE 3.42.

Figure 3.42.

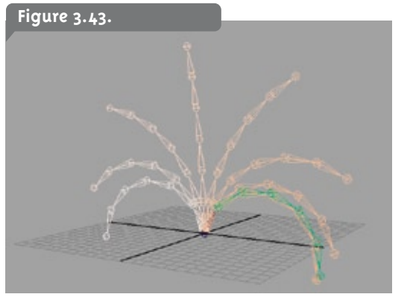


6. Playing back the animation will now show that the rest of the chain joints are rotating exactly as the base joint is FIGURE 3.43. This is not the motion we want. To get the chain joints to rotate in a delayed fashion, we need to make use of some custom float variables. These variables will house the rotation values of joint1 at various time offsets. We can then use these variables to drive the rotations of the chain joints. Add the following code above the existing code.

```
float $minus1;
float $minus2;
float $minus3;
float $minus4;
float $minus5;
```

The dollar sign ($) infront of the variable names is the MEL syntax for storing a variable of any kind. The 'float' variable type is a floating point type or decimal number. We can use these variables to house the rotation values of joint1 at various times throughout the animation.

Figure 3.43.



7. Add the following lines of code to initialize the float variables to something useful.

```
$minus1 = (`getAttr -t (frame -4) ···
        joint1.rz`);
$minus2 = (`getAttr -t (frame -8) ···
        joint1.rz`);
$minus3 = (`getAttr -t (frame -12) ···
        joint1.rz`);
$minus4 = (`getAttr -t (frame -16) ···
        joint1.rz`);
$minus5 = (`getAttr -t (frame -20) ···
        joint1.rz`);
```

Hit the 'edit' button to store the changes into the expression. These lines of code utilize the 'getAttr' mel command. To see a full explanation of what 'getAttr' does, you can look it up in the MEL command reference that comes with Maya under `Help > Mel Command Reference`. Basically what it does, is query the value of an objects attribute so that it can be used in a script or in this case, an expression. You will notice that the 'getAttr' command is followed by '-t (frame -X)'. The '-t' is a flag on the 'getAttr' command that stands for 'time'. It allows you to query an objects attribute value at a specific time. In this case we are querying the value of joint1's rotate Z channel at different times (-4, -8, -12 etc...).

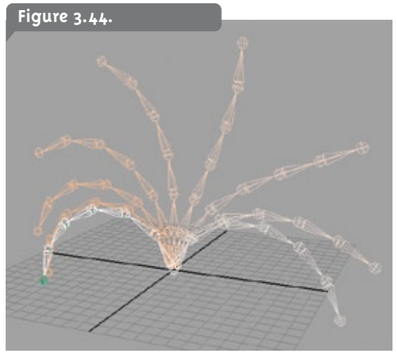
8. Playing the animation now will not show any changes because we need to set the chain joint's rotation channels to be equal to our float variables. Edit the appropriate lines of code to look like this:

```
joint2.rotateZ = $minus1;
joint3.rotateZ = $minus2;
joint4.rotateZ = $minus3;
joint5.rotateZ = $minus4;
joint6.rotateZ = $minus5;
```

Hit the 'edit' button again.

9. Playback to see the effect of our time delay expression on the joint chain FIGURE 3.44. Open exercise3.6_MiddleB.mb to see what this setup looks like at this point.

Figure 3.44.



10. While we are now getting a nice floppy looking motion, it can still be improved by adding a level of randomness to the swaying motion of the base joint. Select the base joint and edit the expression by adding noise to it.

```
float $randomness = noise(time);
joint1.rotateZ = sin(time)*$randomness*30;
```

Hit 'edit' to save the changes. The $randomness variable is initialized to a noise function that is multiplied by time. You can check out 'noise' in the MEL command reference. Basically, the noise function returns a value with a somewhat random nature. We could have used the 'rand' function but noise has the advantage of returning values that aren't completely random, much like the swaying nature of an underwater current. I recommend getting to know all the different math and random functions that are available in MEL. A complete list of them can be found under the 'Insert Functions' menu in the expression editor.

11. The entire expression looks like this:

```
float $minus1;
float $minus2;
float $minus3;
float $minus4;
float $minus5;

$minus1 = (`getAttr -t (frame -4)···
joint1.rz`);
$minus2 = (`getAttr -t (frame -8)···
joint1.rz`);
$minus3 = (`getAttr -t (frame -12)···
joint1.rz`);
$minus4 = (`getAttr -t (frame -16)···
joint1.rz`);
$minus5 = (`getAttr -t (frame -20)···
joint1.rz`);

joint2.rotateZ = $minus1;
joint3.rotateZ = $minus2;
joint4.rotateZ = $minus3;
joint5.rotateZ = $minus4;
joint6.rotateZ = $minus5;
```

12. Playback the scene to see the finished seaweed animation. Open exercise3.6_Finished.mb to see the finished procedural animation. Observe how the noise has affected behavior of the base joint. Also watch as the chain joints wiggle around and appear to behave in a floppy manner.

Now you can try adjusting the expressions to achieve different effects. As practice, try adjusting the base joint expression so that the seaweed looks like it's swaying in a very slow, lazy current. Alternatively, try to make it flop around wildly as though it's caught in a raging river. You can make custom variables and hook them up to attributes on objects to create controllers that can easily adjust the various settings. I have seen this technique used to animate all kinds of things from tails and flapping wings, to stingrays or seaweeds. The possibilities are endless!

MEL Scripting – The Creation of cgTkDynChain.mel:

This last section in the chapter on overlapping action is going to concentrate on the issues involved in building a full production ready script to automate a specific setup. If you haven't done exercises 3.1 and 3.2, I highly recommend reading through them before continuing with this section. Exercise 3.1 demonstrates the exact setup we are going to automate, know it inside and out.

This section assumes that you have a basic understanding of the MEL language constructs and syntax. You should be comfortable with float, string and int variables. This script makes extensive use of procedures, both local and global. You should also familiarize yourself with any MEL commands that you are unfamiliar with by having the MEL command reference open while you read this. Various loops will also be utilized. I've tried to explain as much as possible, but at the risk of losing people in details, I ask that you at least read the MEL lesson in chapter one and watch the included MEL video on the DVD before continuing.

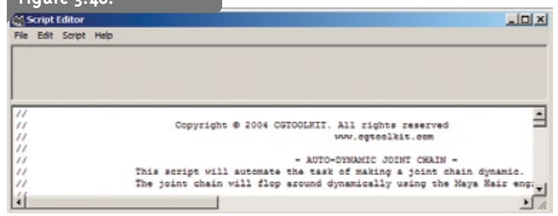
■ Preparation

When you decide to automate a setup task, it is absolutely essential that you have a perfect understanding of the process you are trying to script. Before I create any script, I go through the setup several times and make notes about the specific steps needed to complete the task. These notes can be as detailed or as general as you like, but they will be absolutely necessary to make a plan of attack. This method is not MEL specific but applies to all programming problems. Computer science students are taught to take this concept one step further by writing their programs in a loose language known as pseudo-code. Pseudo code is much simpler than the name might imply. It's simply a way of writing your code in a way that you feel comfortable with. For some, it means using full English sentences to describe any algorithms or procedures. For others, pseudo code looks more like the code it's mimicing. The important thing is that you completely plan out your program before you even touch the keyboard. In the same way that a building needs blueprints, your programs need planning. I can not emphasize this enough!

■ Your Development Environment

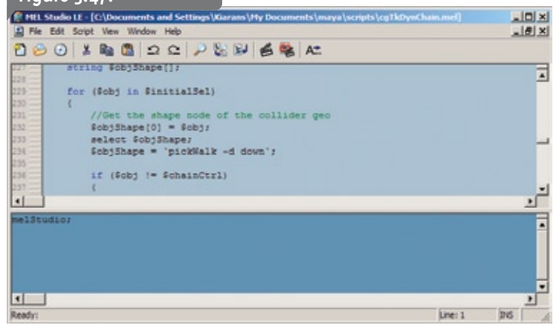
Writing MEL scripts can be done in any ASCII text editor (notepad, xemacs, jedit etc...). Maya's script editor can also be used for writing MEL FIGURE 3.46 .

Figure 3.46.


Script Editor
File Edit Script Help
Copyright © 2004 CGTOOLKIT. All rights reserved
www.cgtoolkit.com
- AUTO-DYNAMIC JOINT CHAIN -
This script will automate the task of making a joint chain dynamic.
The joint chain will flop around dynamically using the Maya Hair eng

That being said, I highly recommend using an integrated development environment or 'IDE'. Personally, I use the excellent MEL editor 'Mel Studio Pro' from Digimation FIGURE 3.47.

Figure 3.47.



There is a lite version of the program that is included on the cd. While Mel Studio LE does have some features disabled, I would still consider it the best option. With Mel Studio, you can edit and execute your scripts within the same window. There is no need to save the file in your external editor and then open and source it in Maya to test your script as you work on it. Another advantage of using Mel Studio over other solutions is the syntax highlighting feature. With syntax highlighting, your scripts will be much more readable and easier to edit and debug. If you choose not to use Mel Studio, be aware that other text editors usually have plugins to support MEL syntax highlighting. Taking the time to setup your development environment will pay off in saved time and a much less stressful work flow.

■ ELF User Interfaces

Usually, I start my scripts by building the user interface for them (assuming there is one). This way, I have a framework with which I can build upon as I add new functionality to the program.

MEL has a great tool set with which you can build user interfaces. In fact, the entire Maya interface is built with MEL. MEL UI's are based on the 'ELF' or Extended Layer Framework paradigm. In practice, this means that your user interfaces are split up into various 'elements'. These elements can be anything from a button, to a slider, to the window itself. Each element has several flags associated with it that you can use to control the specific attributes of that element. Your user interfaces will comprise of a series of MEL commands, the order of which will determine how elements of your UI get arranged on the screen.

■ General Program Structure

In most cases, I like to write my entire script in one, single, text file. If your program is really big (larger than 500 lines) you may want to consider splitting it up into several different files that link to each other via calls to global procedures. This is just good general structural programming practice. For the script file we are constructing in this chapter, I chose to use only one script file. I chose to do this because the entire program is less than 400 lines long, and it is easier to debug if everything is in the same place. Some people go so far as to split up each procedure into it's own script file. This is fine, but it can get confusing when you want your script to be easily distributable. What is important is that you are conscious of the best way to organize your project, especially if you are not the only one that will be using your script.

If you look at cgTkDynChain.mel, you will notice that each procedure is separated by two lines of commented text. This helps to break up the file as you scroll through it to debug. Also notice that the order of the procedures is such that the user interface procedure (named cgTkDynChain) is at the very bottom. This is important because this procedure has a lot of calls to other procedures in the form of button commands. When Maya parses through the script file to execute it, it does so from top to bottom. If it encounters a call to a procedure that has not yet been declared, it will error out and stop executing.

With these structure issues in mind, we can start actually writing the script! Like I mentioned earlier, I like to start with the user interface first to give my script a foundation to build upon.

■ Starting the Script

To start with, open a new script file and give it a header. A header is simply a commented-out portion of code at the top of the script file that contains any pertinent information. I usually includes the following into a script header:

1. The name of the script ie 'cgTkDynChain.mel'.
2. A brief description of the purpose of the script; it's intended use.
3. Directions on how to use the script.
4. Describe the basic procedure for anybody who may have stumbled upon the script and wishes to know, at a glance, what it is that the script actually does.

Here is a look at the header for cgTkDynChain.mel:

```
//
//      Copyright © 2004 CGTOOLKIT. All rights reserved
//                      www.cgtoolkit.com
//
//              - AUTO-DYNAMIC JOINT CHAIN -
//      -This script will automate the task of making a
//       joint chain dynamic.
//      -The joint chain will flop around dynamically using
//       the Maya Hair engine.
//
//     Directions:
//     Select base joint, shift select tip joint and
//      click 'Make Dynamic'.
//     PLEASE NOTE: This script uses Maya Hair to generate
//      the dynamic secondary motion. You must have Maya
//      Unlimited v6.0 or greater to use this script.
```

```
//
//     Basic Procedure:
//     -A CV curve is created through the joint chain.
//     -The curve is attached to a spline ik handle that
//      runs through the chain.
//     -The curve is made dynamic using Maya Hair.
//
```

■ Building the User Interface

The user interface will be contained within it's own global procedure. This procedure is named the same as the script and will be how the user starts the script. Whenever I start a new procedure, I always immediately finish it. This gives myself space to add and test the contents of the procedure. The declaration of our global procedure that will house the user interface is as follows:

```
global proc cgTkDynChain ()
{
       //UI CODE GOES HERE
       print "User interface initialized!" ;
}
```

If you now execute this code, you will notice that nothing happens. The procedure named 'cgTkDynChain' is stored into memory but in order to execute the commands inside it, you must 'call' the procedure. Do so by typing the name of the procedure into the command line and executing it. You should see the results of the 'print' command in the command feedback line as it says 'User interface initialized!'.

Of course, our script is pretty useless right now, so lets start the UI by making a window. Replace the print command with this:

```
window -title "CG Toolkit - Auto Dynamic Joint Chain" ···
dynChainWindow;
showWindow dynChainWindow;
```

Execute the script and call the 'cgTkDynChain' procedure. You will see an empty window pop up with the title 'CG Toolkit — Auto Dynamic Joint Chain' FIGURE 3.48 . The 'window' command created and stored the window into memory. The '-title' flag specified the title that appears in the title bar of the window and the last input, 'dynChainWindow', is the name given to the window so that we can refer to it .

Figure 3.48.

The next line uses the 'showWindow' command. This command takes the name of a window element as input, in this case 'dynChainWindow'. The 'showWindow' command simply shows a window that currently resides in memory. Without this command the window will not pop up.

In order to execute and test your script quickly. Make a shelf button that will quickly call the 'cgTkDynChain' procedure. To do this, type the name of the procedure into the command line, then select it and middle-mouse drag the command to your shelf. A button will be created that will automatically call your script. Now you can execute the script and simply hit this button to test it.

We are well on our way to developing this script. If you are using an external text editor, you may now appreciate the need to have an integrated development environment for creating your MEL scripts. With a shelf button and a copy of Mel Studio LE/Pro, your workflow will be completely streamlined. Always think of ways to make your scripting jobs less stressful.

To get on with the user interface, notice a problem that has already arisen from the few lines of code we now have. If you try and execute this procedure multiple times without first closing the window that pops up, you will get this error:

```
// Error: line #: Object's name is not unique: ···
dynChainWindow //
```

This is because Maya cannot create two windows with the same name. So in order to re-execute your script, you must first close the window that it creates. This is not ideal or even necessary. To solve this problem, we will use a conditional statement to determine if the window named 'dynChainWindow' exists, and if it does, we will close it before attempting to reopen it. This will allow the user to execute the script as many times as they wish without getting any errors. It is also necessary to be able to execute and test the script without hindrance. Add the following 'if' statement above the 'window' command:

```
if (`window -q -ex dynChainWindow`) deleteUI ···
dynChainWindow;
```

The 'if' statement works by determining whether the statement inside the brackets evaluates to true or false, and then executes the 'deleteUI' MEL command if the statement is true. If the statement evaluates to 'false', nothing is done and this line is skipped.

Taking a look inside the brackets, we see the window command again. This time, however, it will not create a window because it is in 'query' mode as specified by the '-q' flag. ELF commands that are in query mode will return values about the queried UI element. In this case, the '-ex' flag returns a boolean value based on the existence of the 'dynChainWindow' window. Check the MEL command reference for a full description of all the different flags associated with the window command.

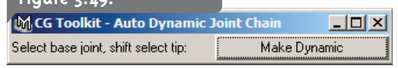
If you now execute the script and hit your shelf button to run it, you will notice that there are no longer any errors. If the window already exists when the procedure is called, the script will delete it and remake it. We are now ready to start adding some buttons!

Add a couple lines to make your entire script look like this:

```
global proc cgTkDynChain ()
{
  // If statement to query the existence of the window
  if (`window -q -ex dynChainWindow`) deleteUI
                                    dynChainWindow;···

  // Command used to create the window
  window -title "CG Toolkit - Auto Dynamic Joint Chain"···
                                    dynChainWindow;
  //Command used to add a layout to the window
  rowColumnLayout -nc 2 -cw 1 175 -cw 2 150;
  // The 'text' command prints text into your UI
  text "Select base joint, shift select tip: ";
  //The 'button' command will add a button to the layout
  button -label "Make Dynamic";

  //Finally, the command used to show the window.
  showWindow dynChainWindow;
}
```

Ensure that the new layout, text and button commands get added before the 'showWindow' command and after the 'window' command. The 'showWindow' command will stay at the bottom of this procedure so that all the elements of the window get created and stored in memory before the window is printed to the screen. Execute the script now, and check out the changes FIGURE 3.49.

Figure 3.49.



Now let's dissect these new commands. The first element in our window is a layout. Layouts are important and you must realize that Maya comes with many different layouts to support a myriad of UI schemes. Read up on them in the MEL command reference so that you are aware of the proper layout to use for any given situation. The 'rowColumnLayout' that we used here will arrange all of the children elements into rows and columns. This layout command has many flags associated with it, but for our purposes we are only going to use two, the '-nc' and '-nw' flags. The '-nc' flag is the shortened form of '-numberOfColumns' and it specifies the number of columns in the layout. So '-nc 2' means there will be two columns in our layout. The '-nw' flag specifies the width of each column and it takes two int values as input. The flag '-nw 1 175' means that the first column in the layout is 175 pixels wide. Similarly, '-nw 2 150' flag means the second column is 150 pixels wide.

It is impossible (and completely unnecessary) to memorize all of the flags associated with every UI MEL command. I always have the MEL command reference open as I build user interfaces so that I can clearly see all of the options available to me with respect to customizing each UI element. The column width values (175 and 150) for the rowColumnLayout were arrived at by trial and error, this demonstrates the nature of building UI's. Lot's of trial and error, testing and re-testing to get the look you want.

If you attempt to create a button without having a parent layout (like our rowColumnLayout), you will get an error. There must be a layout under the window command.

Now that we have our layout setup, and all of the column widths are specified, let's talk about the 'text' and 'button' elements. The 'text' command is pretty straightforward, it prints text into your UI. The button command will create a raised button element that can execute a command when pressed. The '-label' flag specifies the label that will be printed on the button. If you press the button as it is, nothing will happen because we haven't specified a command for it yet. Modify the button like this:

```
button -label "Make Dynamic" -c "dynJointChain";
```

The '-c' flag specifies the command that the button is to execute when pressed. In this case, we have specified the name of a procedure. If you press the button now, you will get the following error:

```
// Error: line #: Cannot find procedure "dynJointChain". //
```

Of course, Maya cannot call a procedure that it does not have! Let's add the procedure. Add the following lines above the 'cgTkDynChain' procedure declaraction (under the header):

```
proc dynJointChain ()
{
       print "Button Works!";
}
```

Execute and run the script. If you now click on the button, you will notice that 'Button Works!' will get printed to the command feedback line FIGURE 3.50 . This confirms that our button is indeed calling an outside procedure. Before we start on the dynJointChain procedure, let's finish up the UI a bit more.

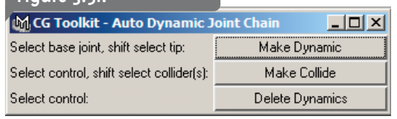
Figure 3.50.

Add the following buttons to the 'rowColumnLayout':

```
text "Select control, shift select collider(s): ";
button -label "Make Collide" -c "collideWithChain";
text "Select control: ";
button -label "Delete Dynamics" -c "deleteDynChain";
```

Execute and run the script FIGURE 3.51. Notice that the text and button elements each get placed in their own cell, inside the 'rowColumnLayout'. This is a nice, well organized, way of showing the user the available options and functionality in the script. The text informs the user of what selection prerequisites are. The buttons are clearly labeled and the intended purpose of each function is clear.

Figure 3.51.



The 'Make Collide' and 'Delete Dynamics' buttons both make calls to other procedures. Without those procedures declared, their execution will result in an error. We will add these functionalities later on. For now, just resist the urge to press them (I know, it's tempting!).

■ The UI Is Done... For Now

Our UI is now to the point where I would leave it alone until the actual function of the script is working. After you get your script's functionality down, you can then worry about adding the finishing touches like error checking and a fancier UI. For testing and development purposes, the UI is now finished. Finally, we can start making the script actually do something!

Let's start by making the 'Make Dynamic' button work. This function of the script is quite complicated and will take a lot of code to accomplish. It must take the currently selected joint chain and setup a spline IK handle with a dynamic, hair-driven spline curve. Then, it must also setup the controller object to give the user a way to adjust the dynamic

properties of the chain.

To do this requires many steps, but to begin with, we must have a way of knowing what joints the user wants to make dynamic. The easiest way I could think of was to have the user select the base joint, then shift select the tip of the chain before executing the 'Make Dynamic' button. In this way, we can specify the exact chain of joints that we want to be controlled by hair dynamics. If you have not already, I recommend watching the included video that demonstrates the finished script in action. This way you know will exactly what we are going for.

Currently, the 'Make Dynamic' button is linked to execute it's own procedure. And as can be seen below, that procedure does nothing of any value, yet.

```
proc dynJointChain ()
{
        print "Button Works!";
}
```

Go ahead and delete the line of code with the 'print' command on it. We know the button works, so this command is no longer needed. Replace the contents of the 'dynJointChain' procedure with the following lines of code:

```
//Store the current selection into a string array.
string $sel[] = `ls -sl`;
//Store the name of the base and end joints into strings.
string $baseJoint = $sel[0];
string $endJoint = $sel[1];
```

These lines of code will determine the joints that were selected as the base and tip joints before the 'Make Dynamic' button was pressed. It is necessary to know this before continuing any further. How can we make the joint chain dynamic, if we do not even know which joint chain was selected?

The first line, above, stores the contents of the current selection into an array of strings. The string variable is an array because it was declared with square brackets ([]). On the same line, the array is initialized to the list of currently selected objects. The command 'ls' will return all objects in your Maya scene. With the '-sl' flag tacked on, it will return all objects in the scene that are selected. There are many uses for the 'ls' command and it has several flags that can be very useful, especially for adding error checking to your script. One final note, before moving on, please note that the 'ls -sl' command is encompassed by back ticks (``). These are important because they cause any command they surround to return the result of the command. So because we used the back ticks, $sel[] will be equal to the returned value of the 'ls' command and not the actual string 'ls -sl'.

Back ticks are very important, please ensure that you fully understand what they do before continuing as I will not explain their use again.

So now that we have a list of the currently selected objects ($sel[]) we can determine the base and tip joints in our chain by finding the first and second selected objects respectively. The string variable '$baseJoint' is assigned the first selected object, and '$endJoint' is given the second. If this does not make sense to you, please understand that arrays are 'zero based' meaning that the first element in our $sel[] array is $sel[0]. If you need more help with using array variables, check out the MEL basics section in chapter one. The Maya help also has excellent coverage of arrays and how to use them.

If you would like proof that these variables are housing the first and second selected objects in your scene, add a couple of print commands after the variable initializations:

```
print $baseJoint;
print $endJoint;
```

Now execute the script. If you now select two things in your scene (a couple polygon spheres will do the trick) and hit the 'Make Dynamic' button you will see the object names printed to the command feedback line. The print command is indispensable as you are testing your script. I use them everywhere to let me know exactly what's going on. Just make sure you delete them or your script will spit out what looks like confusing gibberish to the user. And we get enough of that from our politicians.

■ Building the Curve

Alright, we have come along way and are now ready to start getting into the actual process of making our joint chain dynamic. Remember, we are using spline IK to control the orientation of the joints and we are creating our own curve to use with the IK handle instead of the one it automatically creates. Our curve needs to have a CV at each joint along the length of the chain. This will ensure an adequate resolution so that the joints bend with the curve in an appropriate way.

Creating a CV curve is simple. The command looks something like this:

```
curve -d 3 -p 1 5 6 -p 0 5.898988 -1.84989 -p 0  ...
5.512887 -3.362987;
```

The '-p' flag specifies a point on the curve and is followed by a vector value (three floats separated by spaces). So the point, '-p 1 5 6', will be located at X=1, Y=5 and Z=6 , in the Cartesian coordinate space.

Our curve needs to have it's CVs placed at the world space of each joint. There are a couple of general case issues involved here. Firstly, we do not know where these joints are located (in world space). Secondly, we do not know how many there are. Without these two pieces of data, we cannot construct a curve that will run through any arbitrary joint chain. The only thing we do know is the base and end joint of the chain because the user selected them before executing the script. Fortunately, the base and tip joints are all we need, along with some clever MEL trickery.

So the problem now is:

How do we find the world space of each joint along the user selected chain?

There are many different ways to go about solving this problem. I chose to use a while loop and a couple new variables. The basic algorithm works like this. The while loop starts with the base joint selected. The world space of the currently selected joint is then queried using the 'joint' command before being stored into a float array. After getting the position of the joint, the next joint down the hierarchy is selected and the loop continues. As soon as the loop iterates down the chain to the end joint, it will exit.

Start by declaring a few variables to use in the loop:

```
//Create a vector array to store the world space
//coordinates of the joints.
vector $jointPos[];
//String variable to house current joint being queried in
//the while loop.
string $currentJoint = $baseJoint;
//Counter integer used in the while loop to determine the
//proper index in the vector array.
int $counter = 0;
```

The vector, $jointPos, will store the coordinates of each joint. So $jointPos[0] will be the world space of the base joint, $jointPos[1] will be the second joint, and it will continue in this manner until the end joint.

After declaring the necessary variables, you can write the actual loop:

```
while ($currentJoint != $endJoint)
{
  $jointPos[$counter] = `joint -q -p -a $currentJoint`;
  pickWalk -d down;
  $sel = `ls -sl`;
  $currentJoint = $sel[0];
  $counter++;
}
```

If the conditional statement following 'while' evaluates to 'true', then the commands enclosed in the curly braces are executed and the condition is checked again. This is the behavior of a while loop. The $currentJoint string variable must be equal to the name of the end joint before the loop will exit. This is the exiting condition.

The first command in the loop queries the position of the currently selected joint and stores it into the proper index in the $jointPos array. The second command will pickWalk downwards, which effectively selects the next joint in the chain. The third command will store the current selection into our $sel array. The fourth command then stores the name of the currently selected joint into our $currentJoint string that is used throughout the loop.

Lastly, the $counter integer is incremented with the '++' notation. Incrementing like this has the same effect as simply adding '1' to itself, like this:

```
$counter = $counter + 1;
```

The '++' notation is just a quicker way of typing this commonly used operation of adding one to a variable.

When you are designing loops, you may find it helpful to try it out on paper. Pretend you are going through the same actions as the interpreter and write down the value of each variable as the loop iterates. This can help you find bugs and potential cases where your loop will fail.

If you do that with the loop we just used, you will find that it has a couple problems. Firstly, since the loop pick walks down the hierarchy, we must have the base joint selected before entering the loop. You can do this with a simply command tacked on just before the loop:

```
//Initial selection going into the while loop
select $baseJoint;
```

With that done, everything works fine right up until the end joint. When the $currentJoint string is equal to the name of the end joint, the while loop will exit before adding the end joint's coordinates. We need some way of adding this, knowing that the loop will skip them.

You can do this quite simply by adding these three lines after the loop:

```
//Theses 3 lines store the position of the end joint that
//the loop will miss.
$sel = `ls -sl`;
$currentJoint = $sel[0];
$jointPos[$counter] = `joint -q -p -a $currentJoint`;
```

This could be avoided with the clever use of a do/while loop, but these three lines were an easy addition. Sometimes, finding the most elegant solution takes more time than it's worth.

Now we have the position of each joint stored into a vector array. This is exactly what we need in order to build our curve exactly through the joint chain. What we need to do is build a command that, when executed, will build the proper curve. Since the user's joint chain could contain any number of joints, we must build the command using a loop that takes into consideration any general case.

We will start by using another string variable to house the full command used to build a curve. Remember, the MEL command to build a curve looks like this:

```
curve -d 1 -p <vector>  -p <vector>  .....;
```

In order to build the command, we'll initialize a string to 'curve -d 1 '. Then the loop will add '-p' followed by the appropriate vector value, for each joint.

```
//This string will house the command to create our curve.
string $buildCurve = "curve -d 1 ";
//Another counter integer for the for loop
int $cvCounter = 0;
```

The loop is very simple. It must iterate over the $jointPos array and add each coordinate to the build curve command. The $buildCurve string is concatenated with each iteration until it contains all of the coordinates for every joint. The $counter variable from our first loop will help us determine when the loop should exit because it is equal to the number of joints in the chain.

```
//Loops over, and adds the position of each joint to the
//$buildCurve string.
while ($cvCounter <= $counter)
{                                                    ...
        $buildCurve = ($buildCurve + " -p " +
        $jointPos[$cvCounter]);
        $cvCounter++;
}
```

Again, if you are having trouble visualizing exactly what is going on here, try it out on paper. The loop will gradually build up the command, each time adding another point in the form '-p X Y Z' where x,y, and z are replaced by the coordinates we got from the first loop.

Recall, the $buildCurve string is equal to 'curve -d 1' as we enter into the loop. After the first iteration $buildCurve could look something like this

(if printed out):

curve -d 1 -p 1 2.5 0

After the second iteration, it might print out like this:

curve -d 1 -p 1 2.5 0 -p 2 3.5 0

Then:

curve -d 1 -p 1 2.5 0 -p 1 3.5 0 -p 1 4.5 0

The command string will grow with the loop until the final coordinate is added. We are now almost ready to evaluate this command, to actually create the curve into our scene. To do this, we must first add the terminator to the end of the $buildCurve string, otherwise the command will return a syntax error. Notice that the above printed examples of the $buildCurve string do not show a semi colon (also known as an end terminator) at the end of the command. This line will add that:

```
//Adds the end terminator to the build curve command
$buildCurve = $buildCurve + ";";
```

Now that our $buildCurve command is complete, it is time to evaluate it. The MEL command 'eval', can be used to evaluate a MEL command that is in the form of a string variable. Check the command reference to see more about this command.

```
//Evaluates the $buildCurve string as a Maya command.
//Creates a curve that runs through the joints.
eval ($buildCurve);
```

Go ahead and test out the script so far. Create a series of joints, then select the base, shift select the tip, and hit 'Make Dynamic'. You will see a curve created through the center of the joints! Yay!

■ Making the Curve Dynamic

The hardest part of this script is over. Now it's a matter of applying the hair dynamics and creating the IK spline solver.

Firstly, we need to make the curve dynamic using the command that can be found under the Hair menu, 'Make Selected Curves Dynamic'. In order for the script to know what curve to make dynamic, we need to store the name of the curve into a variable. Let's do this in the same line that we created the curve, by returning the results of the create curve command. Edit the previous line to read like this:

```
string $nameOfCurve = `eval ($buildCurve)`;
```

Now, when we evaluate the $buildCurve command, the name of the curve that gets created is returned and stored into the $nameOfCurve string variable. Making the curve dynamic is now as simple as selecting it and executing the Hair MEL command.

```
//Make curve dynamic.
select $nameOfCurve;
makeCurvesDynamicHairs;
```

When you make a curve dynamic, using Maya Hair, it gets duplicated, and the original is used as a goal shape. It is the duplicate curve that must be applied to the IK spline handle. In order to do that, we must determine what the dynamic curve is named. This seemingly, mundane task is actually somewhat difficult.
The process of finding the name of the dynamic curve, that I chose, utilizes a couple new MEL commands along with some assumptions. At first glance, the following series of commands might look intimidating, but the reasoning is simple. We can safely make the assumption that when Maya created our first curve (using the eval command), it named it using the following convention:

curve#

Where '#' can be replaced by any integer depending on the number of curves that exist in the current scene. If a curve named 'curve1' already exists, then our script will create another curve called 'curve2'.

So, assuming that our curve is named 'curve#' , we can assume that the duplicate, dynamic, curve is named 'curve(#+1)'. To test this out, open a new scene file, create some joints and test the script. The first curve will be named 'curve1' and the dynamic curve will be named 'curve2'. If you execute the script again, in the same scene file, the second dynamic curve will be named 'curve4'.

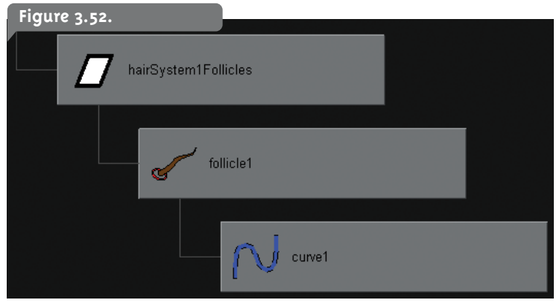
Add the following lines of code to determine the name of the dynamic curve:

```
//Determine what the name of the dynamic curve is
string $nameOfDynCurve;
//The 'size' command returns the length of a string
int $sizeOfString = `size($nameOfCurve)`;
//Add one to the size of the string
$sizeOfString++;
//Use substring to slice a string into a specific portion
$nameOfDynCurve = `substring $nameOfCurve 6 $sizeOfString` ;
$sizeOfString = $nameOfDynCurve;
$sizeOfString++;
//Store the name of the dynamic curve
$nameOfDynCurve = ("curve" + $sizeOfString);
```

The 'substring' command will cut-off the end number on the name of the original string. So if the first curve was named 'curve1', then it will strip everything but the '1' from the name. Then this value is fed into an integer variable ($sizeOfString) and that variable is incremented. The $sizeOfString variable now contains the number that is tacked on to the end of the dynamic curve. Finally, the $nameOfDynCurve string is constructed by concatenating 'curve' with the $sizeOfString variable. Try printing out $nameOfDynCurve to see if it works properly.

When a curve is made dynamic using Maya Hair, an extra DAG node, called a 'follicle' node, is created and can be seen in the hypergraph. The follicle node houses all of the attributes that control the physical properties of the hair curve. We want to create a better interface for adjusting the attributes of the chain, so we are going to connect these attributes to a controller object that will be easier for the user to select and use in the viewport. The original curve (not the dynamic one) is parented underneath this follicle node FIGURE 3.52. We can use this to determine the name of the follicle node, as this will be needed in order to connect to it.

Figure 3.52.




```
//Determine name of follicle node
select $nameOfCurve;
string $nameOfFollicle[] = `pickWalk -d up`;
```

With the original curve selected, we can simply pick walk upwards to find the follicle. Using the backticks (``), we can return the name of the upstream node into a string variable ($nameOfFollicle). If you want to test if this works, try printing $nameOfFollicle and see if it matches the name of the follicle node in the hypergraph.

Before we continue, there is one more node who's name must be determined. The follicle node is connected to a 'Hair System' node. This node contains some of the dynamic attributes that we want to connect to. We are going to use the exact same technique used for the follicle:

```
//Determine what the name of the hair system is
string $nameOfHairSystem;
int $sizeOfString = `size($nameOfFollicle[0])`;
$sizeOfString++;
$nameOfHairSystem = `substring $nameOfFollicle[0] 9 ···
$sizeOfString` ;
$sizeOfString = $nameOfHairSystem;
$nameOfHairSystem = ("hairSystemShape" + $sizeOfString);
```

Again, this is not the most elegant way of finding the name of a node, but it works, and it demonstrates the use of two important commands, 'size' and 'substring'. If you fully understand how to use these two commands, you can manipulate strings to hearts content. Learn them!

If you watched the video demonstrating the use of the finished script, you can observe that the dynamic properties of the chain are housed on a controller sphere that is constrained to the tip of the chain. This controller object is called an 'implicit sphere'. It is a non-renderable sphere object that is normally used as the manipulator for a sculpt deformer. I thought it would do nicely as a controller for our dynamic chains, so that's what I used. I had to use an online forum to find the MEL command that creates implicit spheres. It's amazing what you can learn from the online communities, so use them!

Ranting aside, the command is simply:

```
createNode implicitSphere;
```

So let's add a couple of lines to the script to create a control object and name it intelligently:

```
//Create Joint Chain Controller Object
string $jointCtrlObjArray[];
$jointCtrlObjArray[0] = `createNode implicitSphere`;
$jointCtrlObjArray = `pickWalk -d up`;
string $jointCtrlObj = $jointCtrlObjArray[0];
```

The 'createNode implicitSphere;' command unfortunately returns the name of the shape node, not the transform. Hence, we have to do some pick walking to find the transform node. With the transform node now stored into the $jointCtrlObj string, we can point constrain it to the end joint. This way, the controller sphere will follow the chain as it flops around.

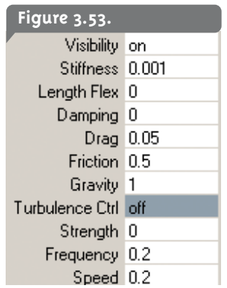
```
//Point constrain the control object to the end joint
pointConstraint $endJoint $jointCtrlObj;
```

With the controller sphere created and constrained, it now needs some custom attributes. There are a lot of attributes to add to the controller, and they all need to be connected to their respective counterparts on either the follicle or hair system nodes. You may notice that I didn't connect to all of the attributes on the follicle and hair system nodes. The attributes that I chose to add to the controller sphere are only those that are most often used. I chose to leave some of the lesser used attributes off the controller sphere just for the sake of keeping it simple and uncluttered.

```
//Add the most used attributes to controller for the
dynamics
addAttr -ln "stiffness" -at double -min 0 -max 1 -dv··
0.001 -keyable true $jointCtrlObj;

addAttr -ln "lengthFlex" -at double  -min 0 -max 1   ···
-dv 0 -keyable true $jointCtrlObj;

addAttr -ln "damping" -at double -min 0 -max 100 - ···
dv 0 -keyable true  $jointCtrlObj;

addAttr -ln "drag" -at double -min 0 -max 1 -dv ···
.05 -keyable true $jointCtrlObj;

addAttr -ln "friction" -at double -min 0 -max 1 -dv·
0.5 -keyable true  $jointCtrlObj;

addAttr -ln "gravity" -at double -min 0 -max 10 -dv ···
1 -keyable true  $jointCtrlObj;

addAttr -ln "turbulenceCtrl" -at bool -keyable true
$jointCtrlObj;
//Unlock turbulence attribute
setAttr -lock on ($jointCtrlObj + ".turbulenceCtrl");··
addAttr -ln "strength" -at double -min 0 -max 1 -dv
0 -keyable true  $jointCtrlObj;
                                                    ···
addAttr -ln "frequency" -at double -min 0 -max 2 -dv
0.2 -keyable true  $jointCtrlObj;
                                                  ···
addAttr -ln "speed" -at double -min 0 -max 2 -dv
0.2 -keyable true  $jointCtrlObj;
```

The 'addAttr' command does the same thing as the `Modify > Add Attribute` window. Check the command reference if you cannot figure out the affect of each flag. Go ahead and run the script now. Check out FIGURE 3.53 to see all of the attributes as they look in the channel box. This is what you should see with the controller selected.

Figure 3.53.

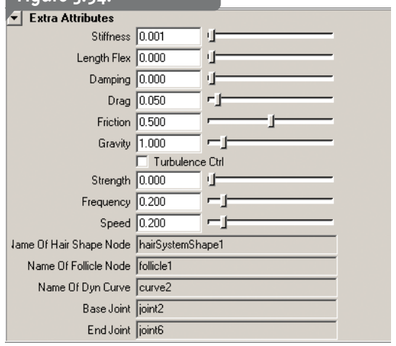


There are a couple more attributes that need to be added. These are non keyable attributes who's only purpose is to house the names of some related nodes. This is needed so that when it comes time to delete the dynamics from the scene file, the controller remembers what nodes are attached to it, and deletes them as needed.

```
//Add special attribute to house the name of  the
//hairSystem node
addAttr -ln nameOfHairShapeNode -dt "string" -keyable ···
false $jointCtrlObj;
setAttr -type "string" -lock true ($jointCtrlObj +
".nameOfHairShapeNode") ($nameOfHairSystem);

//Add special attribute to house the name of  the follicle
//node
addAttr -ln nameOfFollicleNode -dt "string" -keyable ···
false $jointCtrlObj;
setAttr -type "string" -lock true ($jointCtrlObj + ···
".nameOfFollicleNode") ($nameOfFollicle[0]);

//Add special attribute to house the name of the dynamic
//curve
addAttr -ln nameOfDynCurve -dt "string" -keyable false ···
$jointCtrlObj;
setAttr -type "string" -lock true ($jointCtrlObj + ···
".nameOfDynCurve") ($nameOfDynCurve);

//Add special attribute to house the base and end joint
//names
addAttr -ln baseJoint -dt "string" -keyable false ···
$jointCtrlObj;
addAttr -ln endJoint -dt "string" -keyable false ···
$jointCtrlObj;
setAttr -type "string" -lock true ($jointCtrlObj + ···
".baseJoint") ($baseJoint);
setAttr -type "string" -lock true ($jointCtrlObj + ···
".endJoint") ($endJoint);
```

Notice that these 'addAttr' commands have their '-keyable' flags set to 'false'. This will cause the attribute to not show up and clutter the channel box. You can still see these attributes in the attribute editor under the 'Extra Attributes' tab. Execute and run the script to test this out. Select the controller object and look in the attribute editor FIGURE 3.54, the grayed out attributes should be holding the names of the related nodes. Later, when we are programming the 'Delete Dynamics' part of the script, these attributes will come in handy.

Figure 3.54.



Connect the Dynamics

The final part of making our chain dynamic will involve actually creating the IK spline handle, as well as connecting the custom attributes on our controller sphere to the proper nodes.

To start off with, if we want to use the follicle node, we must override the dynamics from the hair system. That sounds scary, but it's just a single checkbox on the follicle's shape node that says 'Override Dynamics'. Before we can set that attribute to 'true', we must find the name of the follicle's shape node. To do that we will use our trusty pick walking again:

```
//Overide the Hair dynamics so that the follicle controls
//the curve dynamics
select $nameOfFollicle;
$nameOfFollicle = `pickWalk -d down`;
setAttr ($nameOfFollicle[0] + ".overrideDynamics") 1;
```

One thing you may notice is that up to this point, hitting 'play' after running the 'Make Dynamic' procedure has resulted in a curve that hangs as though it is nailed at both ends. This is the default setting for dynamic curves using Maya Hair. To change that, adjust the 'pointlock' attribute on the follicle shape node:

```
//Set the dynamic chain to hang from the base joint (not
//both ends)
setAttr ($nameOfFollicle[0] + ".pointLock") 1;
```

If you try it out now, the dynamic curve should hang from the base as it is supposed to. Locking both ends, however, can be useful (swinging bridges, clothes lines etc...). But for our purpose, we want it attached only at the base.

Now, let's connect all of the custom attributes we added to our control sphere. We could have done it before, but our $nameOfFollicle variable is now pointing to it's shape node, which is what we want, whereas before it housed the transform, which is what we wanted at the time. Regardless, now is the time to hook up the attributes:

```
//Connect attributes on the controller sphere to the
//follicle node
connectAttr -f ($jointCtrlObj +".stiffness") ···
($nameOfFollicle[0] + ".stiffness");
connectAttr -f ($jointCtrlObj +".lengthFlex") ···
($nameOfFollicle[0] + ".lengthFlex");
connectAttr -f ($jointCtrlObj +".damping") ···
($nameOfFollicle[0] + ".damp");

//Connect attribute on the controller sphere to the hair
//system node
connectAttr -f ($jointCtrlObj +".drag") ($nameOfHairSystem
+ ".drag");
connectAttr -f ($jointCtrlObj +".friction") ···
($nameOfHairSystem + ".friction");
connectAttr -f ($jointCtrlObj +".gravity") ···
($nameOfHairSystem + ".gravity");
connectAttr -f ($jointCtrlObj +".strength") ···
($nameOfHairSystem + ".turbulenceStrength");
connectAttr -f ($jointCtrlObj +".frequency") ···
($nameOfHairSystem + ".turbulenceFrequency");
connectAttr -f ($jointCtrlObj +".speed") ···
($nameOfHairSystem + ".turbulenceSpeed");
```

The 'connectAttr' command does exactly what the connection editor does. A direct connection between two attributes, nothing else. If you run the script now, your controller sphere will control the behavior of the dynamic curve.

As a style issue, it is always a good idea to clean any unused channels from a controller object. Currently, our controller has translate, rotate, scale and visibility channels, all showing. None of these are needed since the controller is point constrained to the end joint. Let's get rid of them by making them all non keyable, and locked:

```
//Lock And Hide Attributes on Control Object.
setAttr -lock true -keyable false ($jointCtrlObj + ".tx");
setAttr -lock true -keyable false ($jointCtrlObj + ".ty");
setAttr -lock true -keyable false ($jointCtrlObj + ".tz");
setAttr -lock true -keyable false ($jointCtrlObj + ".rx");
setAttr -lock true -keyable false ($jointCtrlObj + ".ry");
setAttr -lock true -keyable false ($jointCtrlObj + ".rz");
setAttr -lock true -keyable false ($jointCtrlObj + ".sx");
setAttr -lock true -keyable false ($jointCtrlObj + ".sy");
setAttr -lock true -keyable false ($jointCtrlObj + ".sz");
```

Finally! We are ready to make the spline IK handle and attach our dynamic curve to the joints. When you are creating a spline IK handle using your own curve (not the one that is automatically created), you must select everything in a particular order. That order is, base joint first, then the end joint, then the curve. So let's do that now:

```
select $baseJoint $endJoint $nameOfDynCurve;
```

With the selection made, we can go ahead and create the IK handle:

```
//Build the splineIK handle using the dynamic curve.
ikHandle -sol ikSplineSolver -ccv false ;
rename ($baseJoint + "ikHandle");
```

The '-ccv' flag is set to false so that the IK handle uses our curve instead of creating it's own, which is what it will do by default. Then, after creation, the IK handle is renamed to give it a unique name for later deletion.

As another style issue, the controller object needs renaming to make it a little more user friendly:

```
//Rename Controller Object
rename $jointCtrlObj ($baseJoint + "DynChainControl");
```

This will name the controller after the base joint of the chain it is controlling. The user will appreciate being able to know which chain the object controls just by reading the name.

Chances are that your dynamic chains will be a part of a larger hierarchy. For example, if the user is using this to control a creatures antennae, then they need a way to connect the chain to the head joint. To get a hair to inherit hierarchal motion, you must parent the follicle node under the parent transform. In the case of our script, it would be nice if the script automatically parented the follicle under the parent of the base joint (assuming there is one). To do so, we must pick walk upwards from the base joint to find the parent, and then parent the follicle node to this transform.

```
//Parent the follicle node to the parent of the base joint
//This will attach the joint chain to the rest of the
//hierarchy, if there is one.
select $nameOfFollicle[0];
//$nameOfFollicle[0] currently contains the name of the
//follicle shape node.
pickWalk -d up;
//After pick walking, we now have the transform node.
string $follicleGrpNode[];
//The follicle is created under a group node.
//Pick walking up will bring us to the top group.
$follicleGrpNode = `pickWalk -d up`;

//Determine parent of base joint
select $baseJoint;
string $parentOfBaseJoint[];
$parentOfBaseJoint = `pickWalk -d up`;

//Parent the follicle into the hierarchy
parent $follicleGrpNode $parentOfBaseJoint;
parent -w $nameOfDynCurve;
```

The script is almost done. As a finishing touch, you can select the controller object so that after the script is run, the user can immediately begin editing the chain's properties:

```
//Select dynamic chain controller for user
select ($baseJoint + "DynChainControl");
```

It is also nice to print out a confirmation to the user that the script has completed. If the script errors out before getting to this print command, the user will know something has gone awry.

```
//Print feedback for user
print "Dynamic joint chain successfully setup!";
```

The 'Make Dynamic' procedure is now complete. The other two buttons require their own procedures. These will be covered in later sections. While following through the creation process, I hope you realized that the way I chose to accomplish this task is only one of a myriad of different ways. In fact, there may be a much easier, more efficient way of doing the same thing. The important thing is not that you necessarily find the most elegant programming solution for every script, but rather that you find one that is quick and easy to program, with a solid logical foundation. Elegance is nice but if, in the process of streamlining your code, you waste time and make it less readable, then you have defeated the purpose. Keep it simple, and try your best to make it solid.

■ Finishing Touches

The more experienced programmers will notice that I did not include ANY error checking in this script. The unfortunate thing about error checking, is that it tends to make your code severely bloated and far more difficult to understand. This is exactly the opposite of what I was trying to accomplish. It is good programming practice to wait until the core functionality of your program is working, before adding any error checking. The techniques used for error checking are usually pretty simple and straightforward. You will use lots of conditional statements, and a ton of print commands for warning users about erroneous selections.

If you are the only one using the script, you may wish to forgo error checking altogether and instead rely on your own know how for making the script work correctly. There is nothing wrong with this. In a production environment, your boss might be happier that you saved time by not writing useless code as opposed to being angry with a script that can potentially generate errors. Check the DVD for a version of the dynamic chain script with full error checking.

■ Collisions

One of the main advantages of using Maya Hair for simulating floppy appendages, is that it can collide with other objects. Consider a creature that has a long tail. If you want to control the tail with Maya Hair, you need a way of preventing the tail from crashing through the ground. Collision objects with Maya Hair can be any polygonal mesh.

This next section will describe, in detail, the process of adding the 'Make Collide' functionality to our script. It is not very complicated, but there are a couple of tricks you must be aware of when scripting something like this.

If we look at the UI code right now, the 'Make Collide' button looks like this:

```
button -label "Make Collide" -c "collideWithChain";
```

The command on the button (-c flag) calls a procedure named 'collideWithChain'. Add the following procedure declaration above the UI procedure.

```
global proc collideWithChain ()
{
        //Collision procedure goes here
}
```

With the procedure declared, the 'Make Collide' button will no longer give an error when pressed. Instead it will execute the commands enclosed by the 'collideWithChain' procedure (which is nothing at the moment).
To find out what commands we need to use in order to setup a polygon object to collide with our hair, we must execute the commands in Maya and watch the feedback in the script editor's history. To try this for yourself, create a polygon sphere to test with. Select a dynamic curve (you can make one with the script if you want), then shift select the sphere. Now before you make it collide, open the script editor and under the 'Edit' menu, select 'Clear History'. This will give you a clean slate to observe exactly what happens when an object is set to collide with a hair. Under the Dynamics menu set, select **Hair > Make Collide**.

You should see something like the following get printed into the script editor:

```
createNode geoConnector;
// Result: geoConnector1 //
connectAttr pSphereShape1.message geoConnector1.owner;
// Result: Connected pSphereShape1.message to ···
geoConnector1.owner //
connectAttr pSphereShape1.worldMatrix[0] geoConnector1. ···
worldMatrix;
// Result: Connected pSphereShape1.worldMatrix to ···
geoConnector1.worldMatrix //
connectAttr pSphereShape1.outMesh geoConnector1. ···
localGeometry;
// Result: Connected pSphereShape1.outMesh to ···
geoConnector1.localGeometry //
connectAttr -na geoConnector1.resilience ···
hairSystemShape1.collisionResilience;
// Result: Connected geoConnector1.resilience to ···
hairSystemShape1.collisionData.collisionResilience //
connectAttr -na geoConnector1.friction ···
hairSystemShape1.collisionFriction;
// Result: Connected geoConnector1.friction to ···
hairSystemShape1.collisionData.collisionFriction //
connectAttr -na geoConnector1.sweptGeometry ···
hairSystemShape1.collisionGeometry;
// Result: Connected geoConnector1.sweptGeometry to ···
hairSystemShape1.collisionData.collisionGeometry //
connectAttr time1.outTime geoConnector1.currentTime;
// Result: Connected time1.outTime to geoConnector1. ···
currentTime //
```

If you ignore all of the commented output, the only actual commands here are the 'createNode' and 'connectAttr' commands. To make a hair collide with a piece of geometry, Maya first creates a geoConnector node, and then makes a series of connections between the geoConnector and both the hair system and the shape node of the geometry. Lastly, the geoConnector is also connected to the scene time, this enables collisions to behave properly as your animation plays back. So in order for our script to be able to connect collision objects, we must replicate this process exactly.

Before going any further with this, let's setup the procedure to recognize the currently selected objects. We will need to split up the selection into several string variables. Start by adding the following lines of code to the procedure:

```
//Variable declarations
string $initialSel[] = `ls -sl`;
string $chainCtrl = $initialSel[0];
```

The first line should be nothing new, we used this exact command in the 'dynJointChain' procedure. The $initialSel[] string is initialized to contain a list of all the currently selected objects in the scene.

The way this script works is, the user first selects the controller sphere at the tip of their joint chain (remember the controller sphere was part of the dynJointChain procedure), then they shift select any number of polygon objects that they want to collide with the chain. When executed, the 'collideWithChain' procedure must recognize that the first selected object is the controller sphere and subsequent objects are all meant to be colliders. For this reason, we are initializing the $chainCtrl string to be the first element in the $initialSel array. So $chainCtrl will be equal to the name of the controller sphere that was selected before the procedure was executed.

The name of the hair system associated with the joint chain must also be known since several connections must be made to it. If you remember from the 'dynJointChain' procedure, we added a custom attribute to the controller sphere that houses the name of the hair system that it controls. Knowing this, we can simply store that name into a variable. The attribute name was called 'nameOfHairShapeNode'. Let's add a couple more lines of code:

```
string $hairShape = `getAttr ($chainCtrl + ···
".nameOfHairShapeNode")`;
string $obj;
string $objShape[];
```

The $hairShape variable is initialized to the value of the 'nameOfHairShape' attribute on the controller sphere object. The other two variables ($obj, $objShape[]) are only needed for the loop we are going to use next.

I chose to use a for-in loop to iterate through the selection and make all the necessary connections for making an object collide. The loop syntax looks like this:

```
for ($obj in $initialSel)
{
        //Commands inside the loop
}
```

There are a couple things that need to be done for each object to make them collide. The basic procedure looks like this:

1. Find the name of the shape node for the object.
2. Create a 'geoConnector' node for the object.
3. Connect all the necessary attributes between the shape, geoConnector and hair system nodes.
4. Print out something to let the user know what has been done.

So, to start off with, we need to find the name of the shape node for the current object in the loop. The following code will do this for us:

```
//Get the shape node of the collider geo
$objShape[0] = $obj;
select $objShape;
$objShape = `pickWalk -d down`;
```

Again, this should be nothing new. We are simply using the 'pickWalk' command to find the name of the shape node for the current object. The name of this shape node is then stored into the $objShape variable so that it can be used later when we are making the attribute connections.
Step two involves making a geoConnector node for the current object. The following code will create the node, and store the name of it into the $nameOfGeoConnector variable:

```
//Create geoConnector node and store it's name into a
//variable
string $nameofGeoConnector = `createNode geoConnector`;
```

To make the geoConnector node actually work, there are a couple of connections that must be made. Remember, earlier, we observed exactly what connections are being made in the background as Maya sets up hair collisions by watching the feedback in the script editor. Well, now we need to use those observations to construct a bunch of commands that will make the proper connections for us. If you replace the names of the objects with there variable counterparts, we are left with this:

```
//Connect all the necessary attributes to make the surface
//collide
connectAttr ($objShape[0] +".message") ···
($nameofGeoConnector + ".owner");
connectAttr ($objShape[0] +".worldMatrix[0]") ···
($nameofGeoConnector + ".worldMatrix");
connectAttr ($objShape[0] +".outMesh") ···
($nameofGeoConnector + ".localGeometry");
connectAttr -na ($nameofGeoConnector + ".resilience") ···
($hairShape + ".collisionResilience");
connectAttr -na ($nameofGeoConnector + ".friction") ···
($hairShape + ".collisionFriction");
connectAttr -na ($nameofGeoConnector + ".sweptGeometry") ···
($hairShape + ".collisionGeometry");
connectAttr time1.outTime ($nameofGeoConnector + ···
".currentTime");
```

There is no magic going on here, I simply took what the script editor spat out and modified it by changing the object names to reflect the variables that we created.

As a last little touch, let's add a print command that will tell the user what object was connected to which joint chain. Users will appreciate clear and concise feedback that reassures them that something they did actually worked. Adding the following will result in a printout for every object that gets iterated over in the for-in loop:

```
//Print output to the user for each connected collider.
print ($obj + " has been set to collide with " + ···
$chainCtrl + "\n");
```

If executed as is, this loop will result in an error. The problem is a simple logical one. The first time through the for-in loop, the $obj variable will be equal to the first element of the $intialSel array. This will cause the script to try and set the controller sphere to collide with itself. To avoid this inevitable scenario, you can enclose the entire contents of the loop in a conditional statement that will check to see if the $obj variable is equal to the $chainCtrl variable. The condition looks like this:

```
if ($obj != $chainCtrl)
{
        //This is where the loop commands should go.
}
```

With this condition added, the loop will always skip it's first iteration and prevent the nasty errors that we had before. The '!=' notation is a boolean operation that means 'not equal to' in English. For a complete list of the boolean operations, see the MEL reference.

The entire procedure looks like this:

```
global proc collideWithChain ()
{
        string $initialSel[] = `ls -sl`;
        string $chainCtrl = $initialSel[0];
        string $hairShape = `getAttr ($chainCtrl + ···
                                ".nameOfHairShapeNode")`;
        string $obj;
        string $objShape[];
```

```
for ($obj in $initialSel)
{

  //Get the shape node of the collider geo
  $objShape[0] = $obj;
  //Make the selection
  select $objShape;

  //Pick Walk downwards
  $objShape = `pickWalk -d down`;

 if ($obj != $chainCtrl)
  {

  //Create geoConnector node and store it's name into a
  //variable
  string $nameofGeoConnector = `createNode geoConnector`;

  //Connect all of the necessary attributes to make the
  //surface collide
  connectAttr ($objShape[0] +".message") ···
  ($nameofGeoConnector + ".owner");
  connectAttr ($objShape[0] +".worldMatrix[0]") ···
  ($nameofGeoConnector + ".worldMatrix");
  connectAttr ($objShape[0] +".outMesh")
  ($nameofGeoConnector + ".localGeometry");
  connectAttr -na ($nameofGeoConnector + ".resilience") ···
  ($hairShape + ".collisionResilience");
  connectAttr -na ($nameofGeoConnector + ".friction") ···
  ($hairShape + ".collisionFriction");
  connectAttr -na ($nameofGeoConnector + ".sweptGeometry")
  ($hairShape + ".collisionGeometry"); ···
  connectAttr time1.outTime ($nameofGeoConnector + ···
  ".currentTime");

  //Print output to the user for each connected collider.
  print ($obj + " has been set to collide with " + ···
  $chainCtrl + "\n");
  }
}
```

The collision feature is now complete. Test it out by setting several objects to be colliders with a joint chain. The loop will handle any number of collision objects, and print out some feedback for each successfully connected collider. Of course, you could add some error handling to make the script more user friendly, but the core functionality is now working.

■ Baking the Joints

The 'Bake Dynamics' button calls a procedure in the script named 'bakeDynChain'. This procedure will record all of the animation on a joint chain in key frames. It does the exact same thing as the menu command `Edit > Keys > Bake Simulation`. Scripting this functionality was not absolutely necessary, but having the ability to bake a dynamic chain with only one click can save a lot time when setting up lots of creatures with lots of appendages.

To use this function, the user simply selects a controller sphere, then hits the 'Bake Dynamics' button. The procedure to bake the keys into the joints goes like this:

1. The script must find what joints are associated with the currently selected chain controller.
2. The script uses a loop to iterate from the base joint to the tip joint in the chain and records the names of all the joints in between into one long string variable.
3. A final string is constructed by concatenating the names of the joints along with the name of the mel command to bake simulations. This string is then executed as a command using the 'eval' MEL feature.

Just like the other procedures, we start by declaring the procedure along with some variables. These variables will help us determine the names of the joints we are trying to bake.

```
global proc bakeDynChain ()
{
        //Declare necessary variables
        string $initialSel[] = `ls -sl`;
        string $chainCtrl = $initialSel[0];
        string $baseJoint = `getAttr ($chainCtrl + ···
                                ".baseJoint")`;
        string $endJoint = `getAttr ($chainCtrl + ···
                                ".endJoint")`;
        string $bakingJoints = "{\"" ;
        string $currentJoint[];
        $currentJoint[0] = $endJoint;
}
```

The $initialSel variable at i equals o will house the name of the controller sphere that the user selected. If you remember from the 'dynJointChain' procedure, we added a couple of custom attributes to the controller sphere that house the names of the base and tip joints in the chain. Knowing this, it is a simple matter of querying these attributes to find the names of the joints we need to bake. The $baseJoint and $endJoint attributes are initialized to their counterpart attributes on the controller sphere.

The $bakingJoints variable will eventually house the entire list of joints that need to be baked. The list of joints to bake must start with '{' and end with '}'. So $bakingJoints is initialized to '{'. Below is an example of how we need to format the list of joints in order to add it to the bakeResults mel command.

```
{"joint4", "joint3", "joint2", "joint1"}
```

The $currentJoint variable simply houses the name of the current joint as needed in the while loop.

```
//Determine joints to be baked
while ($currentJoint[0] != $baseJoint)
{
        $bakingJoints = ($bakingJoints ···
                        +$currentJoint[0] + "\", \"");
        select $currentJoint[0];
        $currentJoint = `pickWalk -d up`;
}
```

The loop iterates from the tip joint up to the base by pick walking through them. Each joint name is then added onto the end of the $bakingJoints string and separated by a comma. When this loop is finished, it will have a list of every joint in the chain from the tip to the joint just under the base, all separated by commas. After exiting out of the while loop, we need to add the last joint as well as the closing curly brace (}).

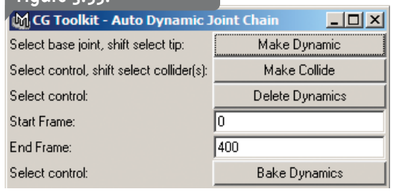
```
//Add the base joint that the while loop will miss
$bakingJoints = ($bakingJoints + $baseJoint + "\"}");
```


Before we can construct the final command and execute it to bake all of the keys, we must also know the frame range that we want to bake. This is specified by a couple of intField UI elements. To add these to the user interface (so that the user can specify a start and end frame), go back to the user interface procedure (named 'cgTkDynChain') and add the following code underneath your buttons.

```
text "Start Frame: ";
intField startFrame;
text "End Frame:";
intField -value 400 endFrame;
text "Select control: ";
button -label "Bake Dynamics" -c "bakeDynChain";
```

Run the script to check out the new fields added to the user interface FIGURE 3.55. When we are done, the user will be able to use these fields to specify the start and end frames for baking.

Figure 3.55.



Back to the 'bakeDynChain' procedure, we can now query the newly added integer fields in the UI to determine the frame range that the user wants to bake keys for. The 'bakeResult' mel command that we are going to use takes the frame range in a specific format. That format is as follows:

```
"0:120"
```

Where 'o' is the start frame and '120' is the end frame. The following code will query the UI elements (the integer fields) and construct a string in the format used above.

```
//Construct frame range variable
string $frameRangeToBake;
float $startFrame = `intField -query -value startFrame`;
float $endFrame = `intField -query -value endFrame`;
$frameRangeToBake = ("\"" + $startFrame + ":" + ···
                                $endFrame + "\"");
```

After constructing the $frameRangeToBake and $bakingJoints strings, we are now ready to concatenate them all together to form the entire command that will bake the animation into our joints.

```
//Concatenate the bake simulation command with the necessary
//joint names.
$bakingJoints = ("bakeResults -simulation true -t " + ···
$frameRangeToBake + "
-sampleBy 1 -disableImplicitControl true - ···
preserveOutsideKeys true
-sparseAnimCurveBake false -controlPoints false -shape ···
true" + $bakingJoints);
```

At a glance, the above statement may look pretty complicated but all I did was copy a 'bakeResults' command from the script editor and replace the frame range and joint list with $frameRangeToBake and $bakingJoints respectively. With the $bakingJoints string now containing the entire command necessary to bake out our joints for the specified range, all we need to do is actually execute this command. For that I used the 'eval' MEL command:

```
//Evaluate the $bakingJoints string to bake the simulation.
eval $bakingJoints;
```

After baking is done, it is nice to print out a confirmation to the user:

```
//Print feedback to user
print ("All joints controlled by " + $chainCtrl + ···
" have now been baked!\n");
```

This will complete the baking portion of the script.

■ Cleaning the Hypergraph

The last procedure in the script will be executed from the 'Delete Dynamics' button. This procedure has the task of finding all of the nodes associated with the currently selected controller sphere and deleting them from the scene. This is a handy feature to help the setup artist clean up the scene after baking all the animation into the joints.

The 'Delete Dynamics' button calls a procedure by the name of 'deleteDynChain' when pressed. Let's declare that procedure along with a couple of variables to house the name of the controller sphere that was selected when the button was pressed.

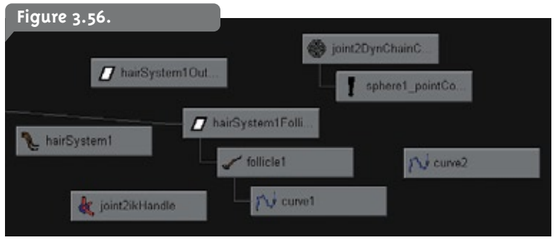
```
global proc deleteDynChain ()
{
        //Declare necessary variables
        string $initialSel[] = `ls -sl`;
        //This string will house the name of the
        //controller sphere
        string $chainCtrl = $initialSel[0];
}
```

Before we start deleting things, let's make a list of all the nodes that need deletion so that we can make a plan of attack.

1. Hair System
2. Follicle
3. Dynamics Hair Curve
4. Hair Output Curves Group
5. Spline Ik Handle
6. Controller Sphere

There are other nodes besides these FIGURE 3.56, but they are parented underneath one of the nodes on this list, so they will be deleted when the parent is deleted. We are only concerned with finding and deleting

Figure 3.56.



nodes that are at the top of their hierarchies.

If you remember from the 'dynJointChain' procedure, we added custom attributes to the controller sphere to house the name of the follicle and hair system associated with it. We can use these now to delete them.

```
//Delete Hair System Node
string $hairSystemName[];
$hairSystemName[0] = `getAttr ($chainCtrl + ···
                     ".nameOfHairShapeNode")`;
select $hairSystemName[0];
$hairSystemName = `pickWalk -d up`;
delete $hairSystemName;

//Delete Follicle Node
string $follicleNode[];
$follicleNode[0] = `getAttr ($chainCtrl + ···
                     ".nameOfFollicleNode")`;
select $follicleNode[0];
$follicleNode = `pickWalk -d up`;
delete $follicleNode;
```

You may wonder why we pick walked upwards in order to delete these nodes. The hair system and follicle nodes are created under their own group nodes. This will delete those as well and prevent any unnecessary group nodes from remaining in the hypergraph.

To delete the dynamic hair curve, we can also query the controller sphere's 'nameOfDynCurve' attribute.

```
//Delete Dynamic Hair Curve
delete `getAttr ($chainCtrl + ".nameOfDynCurve")`;
```

The creation of a dynamic curve will also result in the creation of an empty group node named 'hairSystem1OutputCurves'. To delete this we will just concatenate 'OutputCurves' onto the end of the name of the hair system and then delete that object.

```
//Delete hair output curves node
$hairSystemName[0] = ($hairSystemName[0] + "OutputCurves");
delete $hairSystemName;
```

To delete the spline IK handle, we will construct another string that holds what we know to be the name of the handle. We can do this because we know we named the IK handle after the name of the base joint.

```
//Delete IK Handle
string $baseJoint = `getAttr ($chainCtrl + ".baseJoint")`;
delete ($baseJoint + "ikHandle");
```

Lastly, we can delete the controller sphere itself.

```
//Delete control object
delete $chainCtrl;
```

Just like with the other procedures, it is best to print some feedback to the user.

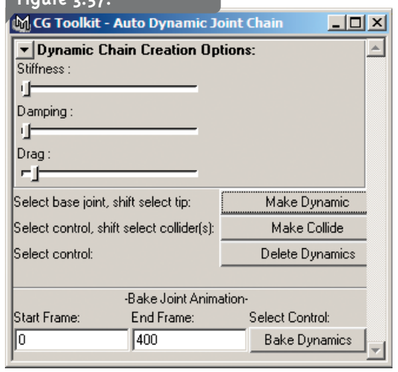
```
//Print feedback to the user.
print "Dynamics have been deleted from the chain.";
```

The clean up feature of the script is now finished. This was the last procedure in cgTkDynChain.mel. All of the features have now been covered and hopefully you have an understanding of exactly how it works.

■ Final Thoughts

With the script's core functionality complete, you are now free to tinker with the user interface and make the front end a little more user friendly FIGURE 3.57. I chose to add some sliders so that you can set some important attributes like stiffness and drag before making the chain dynamic. I also spruced up the interface a bit to make it easier to use.

Figure 3.57.



■ Error Handling

While the script that I have outlined here will work just fine and do it's job well, you must understand that it has not yet been bullet proofed and no error handling has been added. It is best to consider adding error handling to your scripts only once the core functionality has been finished. Error checking can severely reduce the readability of your code and cause headaches during the development process.

The most obvious kind of error that might be encountered during the execution of this script is that of an erroneous selection by the user. For example, in the case of making a chain dynamic, the user has to select a base joint and then the tip before pressing the 'Make Dynamic' button. In the case that the user only selects the base joint and then tries to execute the 'Make Dynamic' procedure, the script will actually cause Maya to freeze by entering into an infinite loop.

These kinds of scenarios can be avoided by adding lots of conditional statements that ensure that the correct nodes are selected before continuing with the execution of the rest of the commands in the procedure.

Check out the finished cgTkDynChain.mel script on the cd to see versions with and without error handling.

■ Chapter Summary:

By the end of this chapter you should not only understand the importance of adding overlapping action to your animations, but also the importance of finding the best way to go about doing it.

Maya's tools are so deep and flexible that they can be combined to create interesting effects in many different ways. This chapter introduced you to five different methods of creating automatic secondary motion. Users who are lucky enough to have a copy of Maya Unlimited were able to see how Maya Hair can be used to create some really cool effects.

If you have also followed along through the MEL script coverage, you should be familiar with how MEL can be used to improve your rigging work flow. MEL can help a talented setup artist to develop new and faster tools to aid them in setting up their creatures.

The next chapter is going to focus on a different and equally interesting topic, Facial Rigging. Creating a believable human face is considered to be one the most difficult tasks in computer animation. Chapter four is a massive section for an equally massive topic. Without further ado, please read on to discover more about this facinating topic.

Chapter 4
The Face

Introduction to Facial Setup and Animation:

This is a huge topic. Facial animation is largely considered to be the most important part of creature and character animation. There are thousands of factors that can contribute to the success or failure of an animated face. Seasoned animators unanimously agree that the human (or caricatured) face displays a level of subtlety and finesse that can take years to master. To gain an appreciation for exactly how complex the face can be, you needn't look further than the mirror. The slightest expression, glare or twitch can send a propagating deformation effect across the entire face and neck. There are 53(!) different muscles in the face that can interact to squash and stretch the skin and fatty tissue layered above. Add into this chaotic mix, a myriad of creasing effects, eyeballs, eyelids, teeth, the tongue and hair and you are left with the most complex system of communication in the natural world.

Assuming your setup is capable of realistically recreating all of the aforementioned effects, you still have the acting issues to deal with. These are the artistic problems in facial animation. The facial animator has to be an actor, in almost every sense of the profession. If the character needs to emote, it is the job of the facial animator to manipulate and contort the face to express to the audience exactly what the director wants. In addition to conveying emotion, a facial animator has to make his creatures speak by syncing the voice track with the movement of the lips and jaw. Lip-sync can be a slow and tedious process. Of course, the term 'lip-sync' is a vast oversimplification since good dialog involves much more than just the 'lips'.

As though things were not difficult enough, you must take into consideration one final problem. With facial animation, the audience will notice things. Since our birth, we humans have been staring at each other's faces every day. When we talk to one another, we look at the face. When we watch the news on TV, we are looking at a face. Some people even spend an hour every day looking at their own face in a mirror (while they put on makeup and do their hair). While some people may never comb their hair in front of a mirror (a fact that doesn't always go unnoticed), they have still experienced enough interaction with fellow humans to become a veritable expert on what a face looks like in every emotion and phoneme. While constructing facial rigs and animation, never forget that your audience is a collection of diligently trained facial experts (ie. regular people). As such, the best inspiration is grounded in reality. Simply buying a mirror for your desk can help you find that tiny detail that takes your animation to the next level.

With all of these seemingly insurmountable technical and artistic issues affecting the final output, it might feel a bit overwhelming. How is one expected to deliver believable, quality facial animation under tight deadlines and to such a discriminating audience? The answer is simplification. This chapter is all about taming the beast. As a creature technical director for next generation games and films, you are responsible for developing believable facial setups that empower the artist. To do so, you will employ a slew of 'hacks' and 'cheats' that combine in a way to fool your 'expert' audience into suspending their disbelief. It is a difficult task, but the rewards are well worth the effort.

In this chapter, I will guide you through the process of using Maya's animation capabilities to give your creatures *believable* facial animation.

Specifically, this chapter will cover:

- Blendshape based facial animation.
- Modeling for animation.
- Defining a facial shape set.
- Constructing a shape set.
- Interfacing with the animator.
- Rigging a 'fleshy' jaw.
- Rigging 'soft' eyes.
- MEL scripting. The creation of Zuckafa's Pose Library.

■ Elements of a Successful Facial Rig

So, in what ways can the setup artist empower the animator? There are several ingredients that combine to create a good facial setup:

1. Diversity: The best setups will allow the animator to reach every possible phoneme and expression. It is not uncommon that a dynamic character will be required to express a full range of 'happy' expressions in one scene, followed immediately by apprehension, then with a hint of fear mixed in. If your facial setup is not diverse enough, the animator may not be able to do his job. Diversity is a bit of a balancing act. You must decide how much control to add without wasting your time with shapes that will never be seen. This is where communication with the animators and directors is of utmost importance. In general, your setup should be capable of at least showing basic expressions and lip-sync.

2. Ease of Use: Giving your animator a list of sixty attributes representing all of the available facial shapes will get you some very angry animators. As the complexity of your setup increases, so should the level of automation. This chapter will demonstrate techniques that can be used to combine controls for different parts of the face to provide maximum control with minimum overhead. A good interface can save your animators hours of work. The interface presented in this book will also help prevent your animator from sculpting an impossible expression. More on this later.

3. Quality: Simple put, how good does the face look? This is the setup artist's place to shine. While creating the shapes for the face, the setup artist must be diligent about sculpting the forms in a believable manner. Good quality facial rigs resist 'pinching' the mesh, even with fairly extreme deformations. In addition to stable, believable shapes, the quality of the facial shapes is largely dependent on the area of influence for each shape. More on this later...

■ Style

The above mentioned issues are fundamental to any kind of facial rig. But for different characters, they can mean different things. The style of your character will determine the exact direction of all the previously mentioned design decisions. There are hundreds of popular styles of animation. Each production will be associated with usually only one. The cohesive nature of styles can enable a single production to look unified across thousands of shots. As a setup artist, you must take into consideration what style is being used and how to apply that to the character's faces.

Most designs can be broken up into a combination of three main styles:

1. Hyper-Realism: You may have heard the term 'hyper-realism' as it gets bandied about in loose-knit circles all the time. The term 'hyper-realism' was coined in the nineteen sixties to describe a rising trend in paintings that looked like photographs. When artists began to break out of the pretentious trappings of modern art, they looked to photographs. Photographs reveal shadow, shape and forms in a 2d format that somehow describes the scene, in artistic terms, better than the human eye. As a result, these 'hyper-real' paintings tended to look 'more real' than the photographs they mimic. In actuality, a hyper-real piece of art is inspired by reality, but the features that make it look real are enhanced and amplified because the artist (either consciously or otherwise) has done so. The end result is something that looks realistic and fantastical at the same time. This is, without a doubt, the most employed style in film effects. Gollum, from the insanely popular Lord of The Rings films, is a prime example of how far hyper-realism can take us. Hyper-realism is harder to achieve in video games (due in large part to their limited polygon counts and rendering capabilities) but the next generation of gaming engines will be attempting to get ever closer. Blizzard Entertainment's Warcraft series showcases some fine examples of hyper-realism as applied to fantasy creatures in video games.

2. Photorealism: Photorealism is perhaps best described as having 'no-style'. The resulting animation is intended to look exactly like a photograph. Photorealism is used in films to seamlessly integrate a digital double into a live action environment. In cases like these, the CGI must look absolutely real. No exaggeration is used to push the animation because this would create a contrast between the different elements of the shot. Digital doubles are probably the main use for photorealism today. This is because it is still much cheaper to employ a real actor to deliver a performance than creating one from scratch in the computer. Photorealism is a great way to protect stunt actors from potentially life threatening stunts. Photorealism can also act as a conduit with which famous actors from the past can come back to life. When creating a facial rig for a photo-realistic character, real-life inspiration is essential. Photorealism is the holy grail of video games, and, unfortunately, we still have a long way to go. Real-time rendering technology is making leaps and bounds but it still has not achieved a photo realistic look. Modern 3d video game engines are advancing at a remarkable pace, but photorealism is still a dream for games.

3. Highly Stylized: Most video games, and some films, fall into this category. Stylized productions afford the most artistic freedom. Characters in a stylized production can range in size and shape with little or no regard paid to whether or not their physicality would even be possible to maintain in the real world. Keep in mind that, although the characters do not have to look possible, highly stylized creatures are often asked to look *believable*. This means that their face must be capable of displaying the same range of emotion as a hyper-real character. While stylized characters are often considered to be the most simple to rig, they can present their own challenges. Difficulty with rigging stylized characters usually results from their bizarre anatomy. As a rigger, you should be available at every step in the production to help avoid character designs that are not conducive to good, clear facial animation.

Of course, these three main styles are not absolute. Many productions employ several different styles throughout both the characters and the environments. Consider Dreamwork's lovable Shrek character. In some respects, he is definitely a highly-stylized character (he's a big, fat, green ogre with tube ears), yet he employs several characteristics of hyper-real characters (sub surface scattering, little fluff on his clothing, beard hairs etc...). In this way, we see that Shrek cannot be strictly fit into any of the three main categories.

When constructing a facial rig, the style of the character must be determined first. Call a meeting with the design team if you must. But before you even open up Maya, make sure that you know what sort of look the director wants.

Blendshape-Based Facial Animation:

There are two main techniques used to create a digital facial puppet, joints and blendshapes. The debate rages on about which is better, but it is this author's opinion that the best way to create a believable facial rig is by utilizing both. As such, this chapter will focus on setting up a face with a blendshape/joint mixture. If your setup is intended for a film/television audience I would *highly* recommend using the mixture method.

The vast majority of modern video game engines support blendshapes (also called morph targets), but not all. Because the fundamental purpose of a 3d engine is to render a game universe at a maintainable framerate, some designers have chosen to forgo supporting blendshapes in favor of the more economical joint-based deformation. This presents a bit of an annoyance to setup artists because their toolset is severely reduced. That being said, these limitations can be overcome, and great looking complex facial animation is still possible using only joints. The problem of creating a limited facial setup with as few joints as possible (to ensure good framerates) can be difficult. For the remainder of this chapter, full access to Maya's joints and blendshapes will be assumed.

On the topic of which is better, blendshapes or joints. There are many people who argue both sides. There are those who argue that a purely joint based facial setup is actually superior to a blendshape setup. They argue that because blendshapes are linearly interpolated, they result in a very mechanical looking face. Joints, on the other hand, deform vertices about a central pivot. The result is a deformation that follows a more natural arc.

Similarly, many TDs argue that *no* joints should be used in a facial setup. They argue that when everything is 'baked' into a blendshape, there is far less overhead and the resulting rig is much liter. In a large production network, lite character files transfer across the network faster, the files open faster and are generally faster to animate. Supporters of blendshapes will say that they offer much more precise control than a smooth bound joint.

Like I mentioned earlier, I like to use a combination of joints and shapes. To understand why I like to use both, you must first understand the argument against blendshapes. A blendshape is simply a displacement of a series of vertices in 3d space. So if you have your character in his neutral pose (staring straight ahead with a blank stare) you can sculpt a duplicate of his head that represents something like a open mouth. The cheek region is pulled down and rotated from the jaw hinge bone to make a nice little shape. The problem with this shape only arises when that shape is applied and interpolated between the default pose. The nasty linear nature of blendshapes is revealed as your creature's jaw looks like it floats down in a straight line with no regard for staying attached at the jaw's hinge.

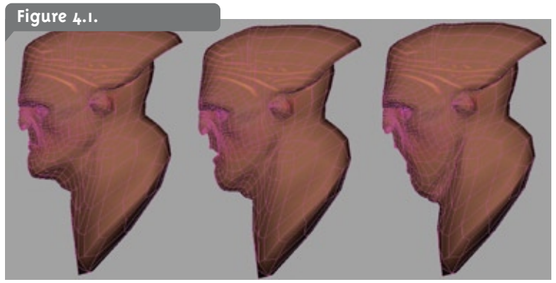
The eyelids are another blendshape problem area. You may have already tried creating an 'eyelid blink' blendshape only to discover that the eyelid intersects through the eyeball at the 0.5 mark. Another nasty effect of linear interpolation.

To make a long story short, joints must be used wherever deformation must occur in an arc motion. That means, that some characters will not need any joints in their face at all. If their jaw is fairly short, the offset at the jaw's hinge from a blendshape will not be noticeable and so a shape may be used. If the characters eyeballs do not protrude from their head at all, a shape may be used without getting any eyelid crashing. You, as a technical director, need to decide on a character by character basis what the best method is. For the character included on the DVD (his name is Zuckafa), I chose to use both joint skinning and blendshapes. This way, you will see how both joints and blendshapes can interact to create an efficient, believable facial setup.

■ Blendshape Theory

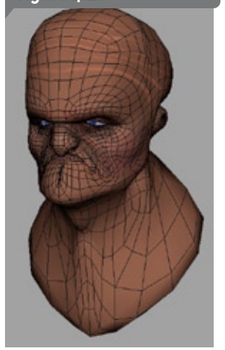
To effectively sculpt blendshapes, you need to have a solid understanding of what Maya is doing in the background when you apply a blendshape deformer. A polygonal mesh is, at it's most basic, a large array of vectors. In non-programmer terms, this means it is a list of points, or a series of vertices in 3d space. Blendshapes simply affect the position of a vertex based on the weight you assign it.

Figure 4.1.



A simple example can illustrate how blendshapes work. Let's say that you have sculpted a 'jawOpen' shape for a character's head. At a weight of 0, the shape will have no effect on the mesh. The jaw will remain closed. At a weight of 0.5, the points being affected will move half of the way to the final shape FIGURE 4.1 . With a value of 1, the head will be fully deformed to look like the 'jawOpen' shape you sculpted. It is important to remember that although the blendshape value's default range is from 0 to 1, you can overload a blendshape by manually typing a value greater than 1 or less than 0 into the channel box. For our jaw example, a negative weight will cause the jaw to close further. It is often very useful to overload a blendshape. Of course, values that are too far from the normal range of 0 to 1 may result in a nasty shearing of the mesh FIGURE 4.2 . It is your job to setup your facial rigs in such a way as to avoid letting the animator adjust a blendshape weight beyond the point where it begins to shear the mesh or look unrealistic.

Figure 4.2.



In addition to overloading, it is extremely important that you understand how an 'additive' shape works. Maya's blendshapes are additive. The additive nature of blendshapes may not be obvious at first, and you will likely need to run into a problem that arises from adding shapes before you fully understand what it means. To try and visualize how Maya 'adds' shapes together, let's consider an example with only one single vertex. Then it should be easy to see how this works on a full face mesh.

Imagine we have our vertex sitting at it's default position. Let's describe this position as a three dimensional vector of the form; (x,y,x). The Cartesian coordinates for our vertex are (0,0,0). Now let's create some simple blendshapes and add them all together to see what happens. Duplicate the single vertex mesh and translate the vertex upwards (in Y) by 5 units. This will create a blendshape. Duplicate the base mesh again and create another shape, this time by translating the vertex upwards by 10 units (in Y) FIGURE 4.5 .

Figure 4.5.

We can describe our newly created shapes as being at (0,6,0) and (0,10,0). Let's call them shape A and shape B respectively. If these were actual shapes, you would select them both and select `Deform > Create Blendshape` in Maya. This creates a blendshape node with a list of all of the target shapes in the channel box. It is from this list in the channel box that we can set the weight values to see how our shapes are behaving.

Set shape A's weight to a value of 1 (100%). As expected, our single vertex will translate upwards to (0,6,0). No surprises. Adding a single blendshape will always result in a completely predictable deformation. The confusion arises when we attempt to add several blendshapes at once.

Figure 4.6.

Now add shape B into the mix by setting it's weight to a value of 1. Our single vertex is again translated upwards, this time to (0,16,0) FIGURE 4.6 . Maya has added both of the deformations together (6+10) to arrive at Y=16. This is contrary to many people's intuition which would say that our vertex should have been translated to (0,8,0) which is the average of the two shapes A and B ((6+10)/2). I want to state this very clearly:

Maya does not mix blendshapes by averaging.

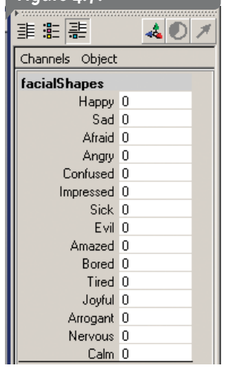
It is this seemingly mundane concept that will allow us to create a facial rig that mimics the way real muscles in the face mix together. Thinking in terms of additive shapes is absolutely crucial to constructing shape sets that mix together without nasty double transformation effects.

Building a 'Bottom-Up' Facial Rig

Now that you fully understand how additive shapes work, I want to impart to you the importance of *why* they work this way. Many people (myself included) started building facial rigs using what can be referred to as a 'top-down' approach. It is probably the most obvious way to construct a shape set when you are a beginner because you needn't think about anything other than what you want the face to look like in each pose.

A 'top-down' facial rig provides the animator with a list of expressions FIGURE 4.7. The animator can use a slider to affect shapes like happy, sad, angry, confused. With lots of experimentation, the animator may even be lucky enough to find a mixture of the expressions that looks appealing when mixed together, that is, without too much shearing. Animating with a setup like this involves meticulously counter-animating shapes against each other. The animator must animate 'happy' down as he increases 'sad'. This results in bouncy facial animation as the entire face must be animated in and out of every shape with a very awkward back and forth type motion. For all intents and purposes, this is simply wrong. I cannot defend a purely top-down approach to facial rigging. It creates more problems than it solves. Prepare to be the most unpopular guy in the studio if you make your animators use a setup like this.

Figure 4.7.



The vastly superior alternative is obviously the 'bottom-up' approach to facial rigging. With a 'bottom-up' approach, the animator is given a bunch of controller objects (usually NURBS curves, but they can be whatever you want). These controllers are connected to shapes that only represent a *single muscle movement*. It is the combination of several of these muscle movements that combine (by addition) to create the individual expressions like happy, sad, confusion etc... The problem of mapping out the face to extract the individual muscle movements will be tackled later in this chapter.

So what about the 'counter-animation' problem you ask? The problem of severe counter animation can be addressed by setting up the controller objects in such a way that they prevent multiple, opposite shapes from being added at the same time. Perhaps the simplest example of how intelligent controller design can prevent counter animating can be seen in the case of the smile shape. A smile widens the entire mouth region and pulls the skin outwards. The direct opposite of this is the pucker shape which squishes the mouth into a 'O' shape and pulls the skin inwards. A smile and a pucker will not mix together, ever. Do not give your animator the option of mixing opposite shapes. If you place pucker and smile on the opposite ends of a controller object, they will never accidentally get mixed. This will be described in detail for all the parts of the face later in this chapter.

In addition to taking care of the counter animating problem, a bottom-up facial rig will also solve the problem of blending from one expression to another. Remember, a top-down facial rig will require that the user ease the entire face in/out of facial expressions. This is completely unrealistic since humans are fully capable of moving their mouth without changing the shape of their eyes. The bottom-up approach separates the face into it's individual muscular components. This enables the artist to sculpt and animate the regions of the face separately. From now on, I will refer to a 'bottom-up' facial rig as a 'muscle-based' facial rig.

Perhaps the only problem that arises with a muscle-based facial setup is that of being difficult to use. Animation may take a very long time if every pose has to be sculpted by the animator from a selection of over twenty controller objects. This is a legitimate concern. It is not uncommon for a scene to incorporate many different facial poses. If the animator has to start from scratch to sculpt each one, the muscle-based system begins to show a weakness. This weakness will be addressed through the use of 'pose-libraries'. The animator can save a facial pose to a file on their hard drive. This file, often called a pose library, can be shared with everybody throughout the production. In this way, the animator can hit a button that places the facial controllers into the proper configuration to represent an expression like happy or confused FIGURE 4.8 . The section at the end of this chapter will cover, in detail, how to use MEL to build a pose library system capable of saving and loading files to the user's hard drive. It is never acceptable to sacrifice time in a tight production schedule. The scenes need to get animated and the animators need the tools to get their jobs done. Pose libraries are a great tool for helping this happen.

One last thing I would like to mention here is the idea of 'combination sculpting'. This is a term that has risen to popularity over the past couple years, most likely because of the major success of the Gollum creature from Lord of the Rings who is said to have utilized the technique. Gollum's facial setup was comprised of over 800 shapes, the majority of which are combination shapes. So what is a combination shape and how do they work?

The muscle-based system that I've presented here can use anywhere from roughly twenty to one hundred different shapes. I guarantee you, this technique can be used to reach every expression that is displayed by the Gollum creature who used over 800 shapes. So why the extra 700 shapes? That is a lot of shapes!

The extra shapes were part of a system of muscle based deformation, just like ours. The exception is that they were used to add extremely minor tweaks to certain poses and combinations of muscle movements to ensure that the final mixture looked perfect. When combining several muscle movements, you may find that the topology of the creature's face has caused the final pose to have a bug in it. This bug may be a minor loss of volume, a shearing vertex, or other minor additive problems. Combination sculpting is a method of taking care of these bugs as they arise.

Maya currently does not have a proper method of dealing with combination shapes. If you feel the need to sculpt a fix for a particular pose, you can write a simple expression to blend the fix in at a certain condition (usually the condition is a series of slider values). The need to use combination sculpting can be largely reduced by constructing your base muscle shapes with special care and attention to being additive. I would recommend you try tweaking your muscle shapes before attempting to add another layer of complexity to your rig via combination shapes.

In the case of a hero creature like Gollum, his face was required to express very complex emotion in full frame at film resolution. For these very special cases, it may be necessary to finesse the deformations via combination sculpting. If you do decide to incorporate a combination shape into your mix, you will likely want a tool that enables you to sculpt the corrective shape on the deformed mesh. There is a script included in chapter 6 that I use for sculpting corrective shapes on the body. It will subtract a shape from a *deformed* mesh and is ideal for creating a corrective shape whether they be for the face or body.

Modeling for Animation:

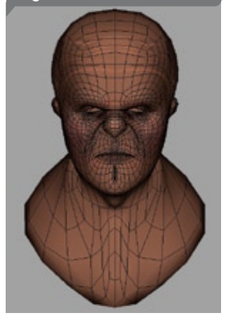
You might be wondering what a lesson on modeling is doing in a book that is decidedly about rigging techniques. The truth is that successful facial animation relies very heavily on having a well constructed mesh. Think of your face mesh as being the base from which all the possible expressions must be extracted. If the mesh is not modeled correctly, no amount of complex rigging will save the day. The mesh is your foundation and if it 'sucks', so will your animation. In addition to covering good mesh construction techniques, I also want to explain the logistics of how to model and setup the non-deforming pieces of the face. These are, specifically, the eyeballs, teeth, tongue, and gums.

Properly modeling the skin is the first step in getting good facial animation. There are several considerations to be made right from the first polygon you (or the modeler) creates. The mesh itself should be one solid piece of geometry that encompasses the entire head, as well as the full neck region down into the top of the chest FIGURE 4.9 . This entire head and neck region will be affected, to some degree, by the blendshapes and smooth skinning on the jaw. Do *not* include the eyes, teeth, tongue or gums as a part of this mesh. These will be separate, non-deforming surfaces, parented into joint hierarchies and should not be smooth bound.

Figure 4.9.

Construction of the skin mesh must be done with careful consideration. If the mesh does not have an adequate resolution, the shapes you build will likely look stiff and boring. In addition to being adequately tessellated, the edges on your mesh must flow in a direction that is perpendicular to the major muscle movements FIGURE 4.10.

Figure 4.10.



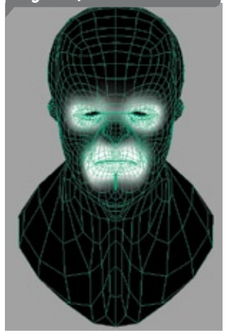
Many modelers prefer a method of blocking out the basic form by starting with a polygon cube. This technique is called *box modeling*. With box modeling, a single, six-sided cube is progressively split, refined and molded into the desired shape. The box modeling technique is notoriously adept at forcing the user to create clean, quadrilateral meshes. While it may seem that box modeling is the easiest, and fastest way to model a face, I would *not* recommend using it. Personally, I have never been able to generate a mesh that flows correctly using the box modeling technique. That is not to say that it is not possible, but you may have better luck with a poly-by-poly method. I find that box modeling techniques are great for quickly fleshing out the basic shape, but the resulting edge loops are usually too rectilinear. In other areas of the body, where edge flow does not need to be nearly as precise, you may find that box modeling is the best method.

When modeling a face that needs to be expressive, I make extensive use of Maya's *Create Polygon* tool. By placing each point and drawing the polygons manually, you can control the direction of the edges with precision. When using this poly-by-poly method, it is often helpful to have a starting place from which you can begin drawing polygons. I like to use the end of a polygon cylinder or a NURBS circle as my starting point.

When drawing polygons, you will find the best layout is one that adheres to the motion of the facial muscles. The face can be broken up into three main areas of action. They are the mouth and the left and right eye sockets FIGURE 4.11. Each of the three regions will deform best with a series of concentric, circular loops that radiate from each region across the face. These circular loops enable the blendshapes to form realistic creases at the corners of the mouth and across the brow. When modeling a face like this using the Create Polygon tool, I like to start with something that is circular. For instance, to model an eye socket, I start with the end of a polygon cylinder, then snap my first row of vertices along the outer edge. This will give you a circular layout from which the rest of the eye region will inherit.

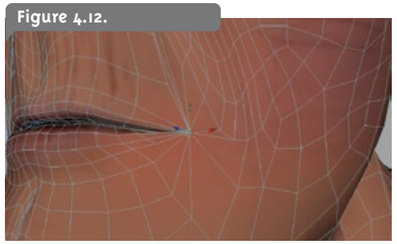
A common problem that arises from modeling the eye and mouth regions of the face separately is how to go about connecting them while maintaining a clean mesh. You may have a couple of nice flowing, circular polygonal surfaces, but if they cannot be combined together, they are useless. There has been much debate over the years as to what exactly makes a mesh 'clean'. The most hardcore modelers will argue that a mesh is only clean if it is completely comprised of only perfect, four-sided faces, or 'quads' as they are called. This is likely a remnant of old patch modeling methodologies when modelers were *required* to construct their patch surfaces from perfect four sided faces. At the risk of being lambasted, I am going to come right out and say that creating a fully quad-based mesh is not nearly as big of a deal as most people make it out to be. That being said, it is still very important to ensure that your faces are split in an intelligent way.

Figure 4.11.



For film and television, where the mesh will be rendered using a software renderer like Pixar's Renderman or mental ray*, a fully quadrilateral mesh is not required. These renderer's are fully capable of drawing n-sided polygons, with no problems. The issues that people have with n-gons is that of surface discontinuities and rigid deformations. If you make the lazy decision of merging several edge loops into one single vertex, you will no doubt create a surface that renders with discontinuities . Try to keep the edges flowing across the mesh equidistantly apart without cramming them into corners and bunching them up FIGURE 4.12.

Figure 4.12.

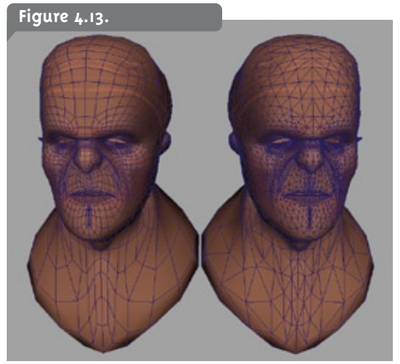


In my experience, a character mesh with the odd five or even six sided polygon will do no harm. Remember, the reason we are using polygons in the first place is because they are much more flexible/faster than NURBS patch models that *require only* four sided faces. The problem of getting rigid deformations from n-sided polygons is something that can be remedied by simply not placing n-sided polygons in places on the body that will undergo alot of deformation.

*mental ray for Maya v3.4 will only render a subdivision surface if it is completely quadrilateral. Non-quad faces in the base mesh will result in a rendering error and the surface will simply be ignored. It can still render a n-sided polygon, but not an n-sided subdivision surface. This annoyance should be fixed in an upcoming version.

For video game engines, the problem of creating a mesh with only quads is even less of an issue. Every current real-time graphics chipset

Figure 4.13.



only renders triangles anyway. Even the Maya viewport (which is powered by OpenGL) tessellates, then renders every mesh as triangles, regardless of how the edges may be drawn FIGURE 4.13 . That being said, the problem of creating surface discontinuities from poorly merged vertices is still there. Anytime your edges cram into a single point, the surface normal at that point will look dark and sloppy.
When you model and merge the separate areas of the face, try to maintain a clear edge flow as much as possible. To sum it up, quads are great, but good edge flow and adequate resolution are far more important.

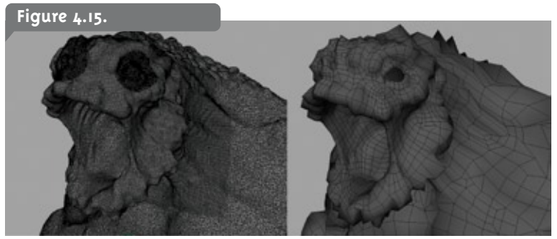
■ Projection Modeling

I want to mention this method of modeling because it is, by far, the easiest way to generate a perfectly clean and accurate creature mesh. Unfortunately, it may require that your studio has a laser scanner (or is willing to adopt a potentially more expensive modeling pipeline). The workflow for projection modeling is quite different.

A physical maquette of the creature is first modeled from clay. This affords the modeler complete freedom to create and sculpt with no artistic hindrances FIGURE 4.14 . The modeler can rough out the form, refine and detail with no regard given to any of the aforementioned technical issues. This will undoubtedly result in creatures that are closer to the concept art designs. Because modeling is such an involved art, clay is still far more intuitive with far fewer mental road blocks that prevent the modeler from freely expressing their vision. This alone may be reason enough to adopt a pipeline that uses clay maquettes.

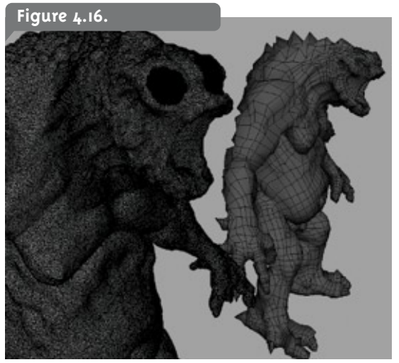
When scanned into the computer, the laser scanning device will create an extremely high resolution triangulated mesh that perfectly represents the creature's dimensions FIGURE 4.15. This kind of accuracy is extremely difficult to attain by modeling from concept art. Unfortunately, the high-res meshes are completely useless to render or animate with because of their excessive resolution. But that is acceptable because the hard work of defining the shape inside the computer is already done. This is where projection modeling comes into play.

Figure 4.15.



Maya 6.0+ has an excellent new feature for projection modeling. Pre-6.0 versions of Maya did not allow the modeler to snap to a polygonal surface (only NURBS). In Maya 6.0, we can now make a polygonal surface 'live'. By making a surface live, Maya will automatically snap the point of your current tool to a point on the surface of the live mesh. This is ideal for the purpose of modeling an animatable mesh from the high-resolution scan data of a maquette. By quickly drawing edges across the scanned surface, the modeler can loft perfect quadrilateral surfaces that conform exactly to the dimensions of the now digitized clay model. Maya is perfectly suited to a pipeline that supports the scanning of clay models. A projection modeled creature mesh will be accurate, cleanly constructed and quick to build FIGURE 4.16 .

Figure 4.16.



There are several scanning facilities that offer displacement mapping services to compliment a laser scanned model. Typically, these facilities will resurface the model for you. The resurfaced model is can then be compared to the high-res scan and the difference is captured into a displacement map. Some scanning facilities are fully capable of recording very minute skin details into a high resolution displacement map. In fact, this technique can also be used for video game characters as long as the game engine supports normal maps. With this technique, the setup artist is free to work with a relatively low resolution mesh, enabling faster interaction and trouble free setup. The finer skin detail is only added at render time or in the rendering engine.

Projection modeling is not limited to use with clay models. The release of Pixologic's Zbrush version 2.0 has opened up a whole new way of modeling detailed forms FIGURE 4.17 . Zbrush is a dedicated modeling program that is perfectly suited to fit into a Maya based pipeline. Pixologic has introduced a new modeling paradigm with Zbrush. Artists can now sculpt polygons almost as freely as clay. You may consider using Zbrush as an alternative to sculpting clay maquettes as it provides a potentially faster turnaround rate. Zbrush also supports the auto generation of displacement and normal maps by comparing high/low resolution surfaces. Your studio may benefit from importing high resolution Zbrush models into Maya and resurfacing them using projection modeling.

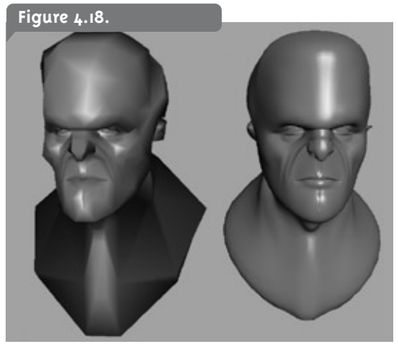
Figure 4.17.

Regardless of whether you use clay or not, projection modeling is probably the fastest way to sculpt clean surfaces. As a setup artist, this is a very important problem to be aware of. The most important thing is that your meshes have the right balance between being well constructed, and being too heavy for your production. Finding this balance is essential.

■ Smoothing

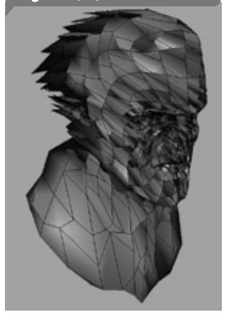
If your creature is intended for a video game, smoothing will not be an issue since most video game creatures are carefully constructed to not waste any polygons. In a film or television production, it is often desirable to add a smooth filter to the mesh before being rendered. Ideally, your smoothing should be high enough to hold up to the closest shot without displaying the faceted nature of the surface FIGURE 4.18. Nothing will break the illusion of digital animation faster than a poorly tessellated surface.

Figure 4.18.



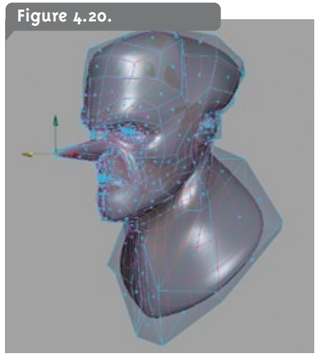
There are two different methods of smoothing in Maya, adding more polygons or using subdivision surfaces. Adding more polygons is as simple as selecting your mesh and executing `Polygons > Smooth`. The channel box will display options for smoothing your mesh. Typically, it is best to use only one or two divisions with a polygon smooth operation. You may find that three divisions is necessary, but probably only for the closest of shots. When setting up a character that needs to be smoothed, it is best to apply the smooth operation after *all other* deformations. The polySmooth node should sit above both the blendshape and skinCluster nodes in the deformation order. If a polySmooth is applied before a skinCluster, the resulting mesh will look something like a porcupine that got pushed through a wood chipper (ie completely screwed up) FIGURE 4.19 . Leave the number of divisions on the polySmooth node at zero while you are animating or setting up the creature. Make sure that the rendering TD's know to set the divisions to one or two before rendering. It may even be handy to have this done with an expression that measures the distance between the surface and the camera and varies the number of smoothing iterations appropriately.

Figure 4.19.



The second method of smoothing involves using a subdivision surface. Subdivision surfaces have the advantage of never showing faceted edges. As a camera is moved closer to a subdivision surface, the mesh is further tessellated to avoid any nasty looking faceting. Subdivision surfaces are very well suited to rendering detailed areas like the face. To setup your surface to render as a subdivision surface, you will need to utilize Maya's wrap deformer. Duplicate your polygonal mesh, then convert it to a subdivision surface `(Modify>Convert>Polygons to Subdiv)`. Then, select the subdivision surface, shift select the polygon cage and execute `Deform>Create Wrap`. This will cause the corresponding vertices on the polygon cage to affect the underlying subdivision vertices. Maya 6.5 has newly improved wrap deformer speeds so that you should be able to interact with your polygonal cage and see the subdivision surface update in near realtime FIGURE 4.20 .

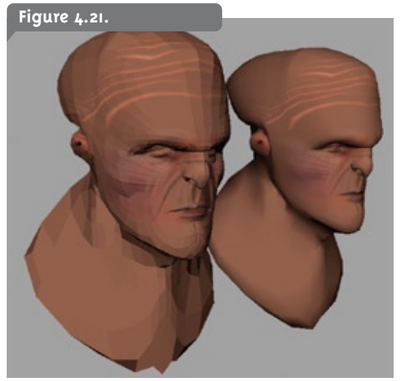
Figure 4.20.



When using subdivision surfaces to smooth your final rendered mesh, be aware that mental ray for Maya v3.4 still does not support the rendering of sub'd meshes with non quad faces. That means that you will either have to ensure that your mesh is constructed from purely quadrilateral faces, or find a different rendering solution. Also, be aware that any sort of smoothing operation will add significant overhead that will be immediately noticeable in the form of bloated rendering times. Never over-smooth a surface.

It should be noted that while video game meshes will not benefit from a smooth operation or a subdivision surface, they can still have their *normals* smoothed. A surface normal is simply a normalized vector that represents the direction that the surface is pointing in. This helps the rendering engine determine how to draw specular highlights across the surface. If the normals are not smoothed across the surface, the specular highlights will look like a sharply cut diamond. To smooth the normals on a polygonal mesh, select `Edit Polygons> Normals>Soften/Harden`. In the option box, select 'All Soft(180)' and hit 'Soft/Hard'. The effects of this operation will be immediately visible in the viewport FIGURE 4.21 .

Figure 4.21.



■ Model Preparation

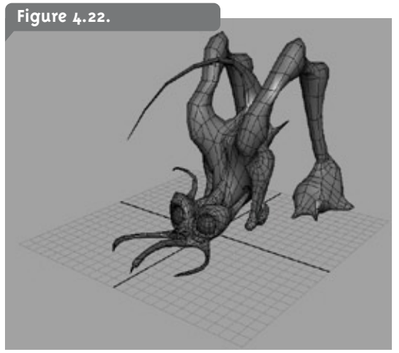
The final task in preparing your model for rigging is the cleanup phase. After the mesh has been completely constructed, the UV's are perfectly laid out and the edge loops are looking good, you must check the integrity of the model. If a broken mesh gets past your inspection and further down the pipeline, you will likely need to fix and re-bind the mesh. This is a complete waste of time and effort. While solutions exists for hacking together minor fixes after being smooth bound, they usually add complexity in the form of extra node history. It is well worth the couple of minutes that it takes to ensure that your mesh is solid. These tests and fixes are relevant to the entire body of your creature, not just the face.

The first thing you, as the setup artist, must determine, is if the mesh is at the proper scale. Personally, I always use real-world scale. So if my creature is supposed to be six feet tall, I measure the mesh using Maya's distance tool `(Create>Measure Tools> Distance Tool)`. By default, Maya uses centimeters as it's base unit. This means that a six foot tall creature should measure 182.88 units with the distance tool. Scale is extremely important whenever your production involves multiple characters that need to interact. Imagine a scene with ten characters imported with different scales. This is a logistical nightmare that can be avoided by simply laying down some standards at the beginning of the production. If your production will be utilizing dynamic simulations (ie. cloth), the dynamics TD's will appreciate having the character in real world units.

It is likely that you will be setting up a creature's face separate from the rest of it's body. In this case, it is important to ensure that the face is scaled appropriately. It is always easier to scale a mesh *before* you start rigging it.

Most productions have the creature's feet resting on the plane where y equals '0' FIGURE 4.22 . This is known as the ground plane. You may find it useful to temporarily snap the pivot of the mesh to a vertex on the underside of the foot. With a positioned pivot point, you can simply point snap the mesh to the origin and the creature will be perfectly standing on the ground plane. Rotate the model so that it faces down the positive Z axis. This is the world alignment that most productions use.

Figure 4.22.



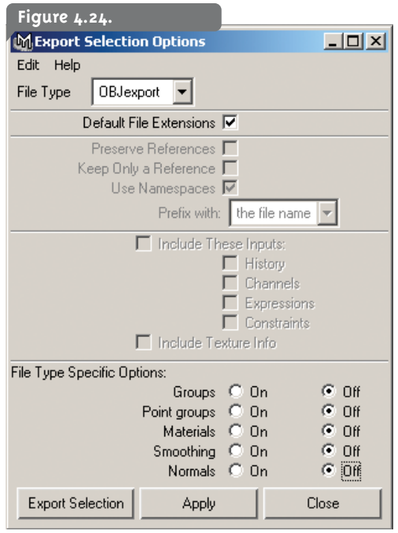
After setting the scale of the mesh, you must ensure that it is not under any groups nodes. Check in the hypergraph to make sure it is a parent of the world. Un-parent the mesh if necessary. Having done all this translating, rotating and scaling, the model will display garbage values in the channel box. Clean this up by freezing the transforms (will set the transform values to equal 0, `Modify> Freeze Transformations`) and center the pivot.

Maya has some great tools to help with checking the integrity of your polygonal meshes. Giving the model a quick look can catch 99% of the problems you may encounter. Smooth the model and apply a blinn shader (blinns have very pronounced specular highlights). Orbit around the smoothed model and study the surface. Pay special attention to the way the specular highlights glide across the faces. If the highlight seems to jump or glitch, you have likely found a problem with the mesh FIGURE 4.23 . Things that can cause your mesh to screw up include, having two un-merged vertices directly on top of one another, having non-manifold geometry, or it could be a problem with the normals. Maya's polygon cleanup tool will check for non-manifold geometry and you can use the Edit `Polygons> Merge Vertices` tool to take care of a stray vertex. Try rendering it from every angle and make absolutely sure that there are no problem areas before continuing.

Figure 4.23.

Maya includes support for the excellent .obj file format that can be used for creating clean mesh files. An .obj file is a very basic format for the import/exporting of mesh data. As a final way of ensuring that your creature file does not have *any* extra nodes in it, you can select the mesh and export it as an .obj file. Turn 'off' all of the .obj save options to save out a completely clean file FIGURE 4.24 . If you cannot see an .obj option under the export settings dialog, you may need to load the .obj plug-in by selecting 'objExport.mll' under Maya's plugin manager. Obj files are lifesavers, never leave home without.

Figure 4.24.



Defining A Facial Shape-Set:

This section will describe the last bit of preparation that needs to be done before you can start actually building your facial rig. A shape set is necessary to help you define exactly what your character's facial rig must be capable of displaying.

It may be tempting to skip over this preparation phase and dive right into building some shapes. Please try to resist this as I can guarantee it will only lead to disaster. A shape set must be completely fleshed out before any work is started. Defining your character's shape set will definitely take some time. This section of the chapter is designed to get you the essential information needed to speed up this process. The shape set that we build in this chapter can be used as a blueprint to construct a facial rig that is capable of extremely diverse, accurate and appealing facial deformations.

■ Understanding the Character

All shape sets are not created equal. In the world of character rigging, artists are always trying to devise an all-encompassing solution to every rigging problem they tackle. This constant pursuit of 'the perfect rig' can, at times, be detrimental to the production. In a production environment, where many character's need to be rigged with facial animation capabilities, it is important to determine the amount of detailing that must be done per character. Each separate character, in every different production, will have different needs. Your shape set should reflect these specific needs without being too detailed to the point of being wasteful.

In a film or television production, you will likely have the advantage of knowing exactly what shots your character will be involved in. Any good production should have a myriad of different avenues to explore in order to determine exactly what the character should be capable of expressing with his/her facial system.

Start at the script or storyboards and highlight all of the shots that the character is involved in. Take special notice of any dialog, expressions or actions that the character must be capable of displaying. Assembling a simple list of the expression requisites is the first step in defining a shape set for your character. Take, for example, the famous scene from Blue Sky's feature animation, Ice Age, where the squirrel-esque character (Scrat) is bouncing around trying to find his acorn in the snow. While this scenario does not necessarily imply that Scrat should be capable of complex lip-sync, it does require that he can emote (because he is *very* concerned about saving his acorn). As such, even this secondary character, with no dialog, must have a facial animation system with some degree of expressive capability. The amount, and complexity of that expressiveness, is determined by you, the setup artist.

Because the facial rig will likely be used by other people, it is important to include their input during this process of defining a character's facial needs. Call a meeting with the animators and the animation director if you have to. Just make sure that your facial rig will be capable of communicating the intentions of the script and storyboards.

More often than not, a large percentage of the creatures in your production may not need to talk at all, or even express more than the bare minimum. Consider a creature who may only be seen from a distance, or for a brief moment. Creating a full-fledged, complex facial system for a creature like this is wasteful and useless. Your time could be much better spent improving the creatures that *do* need complex, emotive setups.

Taking all of this into consideration will likely result in a slew of different setups. Character A's facial system can be very different from character B. This is fine for a film or television production, but next generation video games may need a more uniformly consistent method. The character's faces may need to be animated 'on-the-fly' as in the case of next generation video game productions. Where the face will not be directly animated by a person, you will likely need to construct a generic facial rig with only the controllers that are programmed into the game's dynamic animation engine . In this case, the facial system will need to be fairly generalized and capable of the widest range of expressions.

The shape set used with our character, Zuckafa, will give the animator (or game engine) a very wide range of possible expressions. Depending on the needs of your production, you can scale this generic shape set to match your needs. A complex film creature may need extra shapes to reach a particular pose in a specific shot. For a simple game character, our setup may be overkill. While you will likely want to scale the shape set in this book to match your production needs, it is a good example of a fairly diverse, generic setup.

■ The Human Face:

Once the requirements for your character have been settled, the task of breaking up the needed expressions into individual shapes can begin. Remember, for this book, we want to build a generic setup that is capable of the widest range of expression. This way we can pare our shape set down for less complicated characters. We can also use this generic model as a strong foundation with which a more complicated facial system can be built upon (if needed). Regardless, the shape set must be largely all-encompassing and fully capable. In addition to being diverse, remember that we want our system to be a muscle-based (bottom-up) facial system.

The problem of cutting up a face into manageable, single muscle movements sounds daunting, and it is. Fortunately, for us CG artists, this task has already been done for us. There has been much research on exactly this topic, the majority of which has been done by a researcher at the Univeristy of California, Dr. Paul Ekman. Dr. Ekman is a world renowned psychologist who's interest in the human face has erupted into a bevy of publications over the years. His pioneering work on nonverbal behavior and emotions as they apply to the human face is extremely valuable for our purposes. Modern CG artists can thank Ekman's research for quantifying the muscular movements of the face.

Dr. Ekman has several publications that describe his different finding with regards to the mechanics of facial deformation. Of particular interest to those looking for a comprehensive collection of possible facial movements, is Dr.Ekman's Facial Action Coding System or FACS. Originally published in the nineteen seventies, FACS has provided professionals, from many different fields, with a very detailed explanation of exactly what the human face is capable of. FACS is actually a fully realized training system that comes complete with manuals, videos and pictures. The entire package can be purchased online from www.face-and-emotion.com.

For our purposes, all we really need from the FACS is a list of available muscle-movements in the human face. Fortunately, this much of the FAC system has been made available to the public for free. Specifically, we are interested in the 'Action Units' that are described by the facial coding system. These 'action units' are our way of breaking down the face into workable portions. Dr.Ekman's facial action units describe the precise movement of each individually recognizable muscle group in the human face. We can use these to further narrow-down our shape set to arrive at a final list of individual shapes and motions that can then be added together (remember, blendshapes in Maya are additive) to arrive at all the expressions that our character will need.

Knowing the action unit system can be very helpful for a facial setup artist. There are a total of forty-four different action units as described in the facial action coding system. They are described as individual units and are named from AU-1 to AU-65. The combination (or addition of blend shapes in Maya) of different action units can describe different expressions like happy, sad or angry. Using this method, different facial expressions can be expressed in terms of the action units they involve. For example, looking 'angry' could be described as AU-4 + AU-9 + AU-10 FIGURE 4.25 . If AU 4, 9 and 10 are engaged, the face is snarling, pulling the top lip upwards, and compressing the brow region. This expression will most likely be interpreted as looking angry, maybe even evil.

So by adding various intensities of the forty-four different action units together, we can express any possible emotion in terms of AU-X + AU-Y + AU-W... where A,Y and W are some number representing a different action unit. Of course, complex facial expressions may involve more than just three different action units. You must also recognize that not all action units can be combined in full strength. In the case of our facial rig, if two conflicting action units are combined, the result will likely be either a very strange looking facial expression, a sheared mesh, or a loss of volume in the face. A carefully constructed shape set can reduce the chances of getting a poor looking combination, but ultimately, the animator must decide what looks incorrect, and how to properly blend action units to generate realistic expressions.

Figure 4.25.

The entire list of action units (according to Dr.Ekman's Facial Action Coding System) includes forty-four different units. They are arbitrarily numbered from zero to sixty-five. The following table (on page 98) describes all of the different action units, and exactly what muscles are involved. Remember that many of the action units are symmetrical and have corresponding left and right versions. The task of breaking these units down into individual left/right shapes is tackled later in this chapter.

While it is not necessary to memorize all of the action units, knowing what the action units are can help you communicate in the language of expression. Some professional animators/riggers often use the action unit language when they are describing a very precise expression. When there are multiple people working on a single facial setup, using the action unit language (ie. AU-4 + AU-9 + AU-10 = angry) will help everybody to communicate with clarity.

All 44 Action Units from the Facial Action Coding System:

Action unit	Name	Description of action	Muscles Involved
1	Inner Brow Raiser	Pulls the inside of both the left and right brows upwards. FIGURE 4.26	Frontalis, pars medialis
2	Outer Brow Raiser	Pulls the outer region of either the left or right brow upwards. FIGURE 4.27	Frontalis, pars lateralis
4	Brow Lowerer	Pulls either the left or right brow downwards. FIGURE 4.28	Corrugator supercilii, Depressor supercilii
5	Upper Lid Raiser	Pulls the upper eyelid open. Great for creating a surprised or excited look. FIGURE 4.29	Levator palpebrae superioris
6	Cheek Raiser	Raises the cheeks underneath the eyes upwards. This is very close to squinting. FIGURE 4.30	Orbicularis oculi, pars orbitalis
7	Lid Tightener	Raises the lower eyelids. Can create some great expressions (recall the scene from Ice Age where the Scrat character's bottom eyelid begins to quiver with fear, classic). FIGURE 4.31	Orbicularis oculi, pars palpebralis

Figure 4.26.

Figure 4.27.

Figure 4.28.

Figure 4.29.

Figure 4.30.

Figure 4.31.

Action unit	Name	Description of action	Muscles Involved
9	Nose Wrinkler	Pushes the skin on either the left or right side of the nose up. Creates lots of wrinkles and looks like a snarl. Every monster needs a good snarl. FIGURE 4.32	Levator labii superioris alaquae nasi
10	Upper Lip Raise	Pulls the upper lip up, usually causing the upper teeth to show. Can look aggressive. FIGURE 4.33	Levator labii superioris
11	Nasolabial Deepener	The nasolabial fold is the crease that runs from the nose, to the corner of the mouth. This movement deepens that crease. FIGURE 4.34	Zygomaticus minor
12	Lip Corner Puller	Tugs the corner of the mouth outwards and up. This is heavily activated in a big smile or a laugh. FIGURE 4.35	Zygomaticus major
13	Cheek Puffer	Like the lip corner puller, the cheek puffer tightens the lip corners. Useful for sealing the mouth to 'puff' it out. FIGURE 4.36	Levator anguli oris (a.k.a. Caninus)
14	Dimpler	Aids mastication (grinding/chewing) and can create dimples in the cheeks. Intensity varies greatly between faces. Zuckafa's face does not dimple. Character's with fat cheeks are much more likely to dimple. Also, dimples are often thought to be 'cute', this is something that would not really suit Zuckafa's character.	Buccinator

Figure 4.32.

Figure 4.33.

Figure 4.34.

Figure 4.35.

Figure 4.36.

Action unit	Name	Description of action	Muscles Involved
15	Lip Corner Depressor	Pulls the corners of the mouth downwards. This is a major part of frowning. Causes major creasing. FIGURE 4.38 Figure 4.38.	Depressor anguli oris (a.k.a. Triangularis)
16	Lower Lip Depressor	Pulls the entire lower lip down, usually enough to bear the lower teeth. Used in many expressions. FIGURE 4.39 Figure 4.39.	Depressor labii inferioris
17	Chin Raiser	Pulls the skin around the cheek upwards. Usually creates very chaotic wrinkle patterns across the chin. Most useful in pity expressions. Zuckafa does not use this action. Because this action requires a lot of geometry to create the wrinkle patterns, it is best used in only very extreme cases.	Mentalis
18	Lip Puckerer	Squishes the lips into a kissing formation. FIGURE 4.41 Figure 4.41.	Incisivii labii superioris and Incisivii labii inferioris
20	Lip Stretcher	The opposite of lip puckerer. The lip stretcher is very important for 'e' sounds in lip-sync. Also part of a smile. FIGURE 4.42 Figure 4.42.	Risorius w/ platysma
22	Lip Funneler	Pushes the lips into the shape of an 'o'. Oddly enough, it is used to make the 'o' sound as well. FIGURE 4.43 Figure 4.43.	Orbicularis oris

Action unit	Name	Description of action	Muscles Involved
23	Lip Tightener	This is the action of pursing your lips. Squishing them together without pushing them outwards. Creates a great deal of wrinkles. FIGURE 4.44 Figure 4.44.	Orbicularis oris
24	Lip Pressor	Same motion as the lip tightener but extreme enough to cause the lips to curl inwards. Useful for the 'm' sound. FIGURE 4.45 Figure 4.45.	Orbicularis oris
25	Lips Part	Relaxation of the lip muscles causing the lips to hang open. Useful for getting a 'dumb' expression, like Bubba from Forest Gump. FIGURE 4.46 Figure 4.46.	Depressor labii inferioris or relaxation of Mentalis, or Orbicularis oris
26	Jaw Drop	This action rotates the jaw joint to open the mouth. Used in many expressions and throughout lip sync. FIGURE 4.47 Figure 4.47.	Masseter, relaxed Temporalis and internal Pterygoid
27	Mouth Stretch	Mouth is opened far enough to stretch the muscles around the mouth enough to inhibit their movement. Lip sync is impossible in this extremity. FIGURE 4.48 Figure 4.48.	Pterygoids, Digastric
28	Lip Suck	A more extreme version of the lip pressor movement, this pulls the lips completely inside the mouth. FIGURE 4.49 Figure 4.49.	Orbicularis oris

Action unit	Name	Description of action	Muscles Involved
41	Lid Droop	This action is the result of a lazy top eyelid. Lip droop is the first sign that your character is smoking marijuana or is really tired. FIGURE 4.50	Relaxation of Levator palpebrae superioris
42	Slit	Relaxation of the top eyelid while pulling up the bottom lid resulting in a tiny slit to look through. Eyes look very tired. FIGURE 4.51	Orbicularis oculi
43	Eyes Closed	Top eyelid is pulled all the way down. Essential part of blinking. FIGURE 4.52	Relaxation of Levator palpebrae superioris; Orbicularis oculi, pars palpebralis
44	Squint	Skin around the eyes is pulled inwards. Creates crows feet creasing patterns. Useful in many expressions. AU42 + AU44 = trying to see something far away. FIGURE 4.53	Orbicularis oculi, pars palpebralis
45	Blink	A very quick close/open action of the eyelids. One could write a whole chapter on animating blinks. FIGURE 4.54	Relaxation of Levator palpebrae superioris; Orbicularis oculi, pars palpebralis
46	Wink	Closing left/right eyelids while keeping opposite eyelids open. Usually not a natural motion, very forced. Partial winks can make your character look drunk (think Homer Simpson). FIGURE 4.55	Relaxation of Levator palpebrae superioris; Orbicularis oculi, pars palpebralis

Figure 4.50.

Figure 4.51.

Figure 4.52.

Figure 4.53.

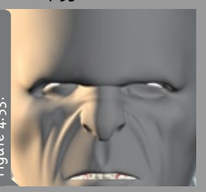


Figure 4.54.

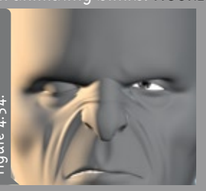
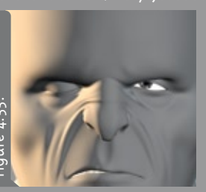


Figure 4.55.



Action unit	Name	Description of action	Muscles Involved
51	Head Turn Left	Entire head pivots left. This action involves very few neck vertebrae and should be completely separate from any shoulder movement. Head can rotate left about 70-90 degrees from straight. FIGURE 4.56	Skeletal
52	Head Turn Right	Same as 51 but opposite. FIGURE 4.57	Skeletal
53	Head Up	Head rotates upwards. The chin is raised resulting in a defiant, possibly proud or arrogant posture. FIGURE 4.58	Skeletal
54	Head Down	Head rotates downwards. The head is lowered, resulting in a passive, obedient or possibly mournful posture. FIGURE 4.59	Skeletal
55	Head Tilt Left	Head rotated on the horizontal axis to the left. Head tilts can be used in conjunction with tons of facial expressions. Often used to convey playfulness, interest or lack of understanding, depending on the context. FIGURE 4.60	Skeletal
56	Head Tilt Right	Same as left tilt, but opposite. FIGURE 4.61	Skeletal

Figure 4.56.

Figure 4.57.

Figure 4.58.

Figure 4.59.

Figure 4.60.

Figure 4.61.

Action unit	Name	Description of action	Muscles Involved
57	Head Forward	Entire head is pushed forward by engaging lower cervical vertebrae. The head forward action can be used to show disbelief or interest. People often do this when they are examining something. FIGURE 4.62	Skeletal
58	Head Back	Entire head is pushed back by engaging lower cervical vertebrae. Head back can be used to compliment surprise (being 'taken aback'), disgust or stunned expressions. FIGURE 4.63	Skeletal
61	Eyes Turn Left	Eyeballs rotate in their sockets to look to the left. The direction of the eyes in relation to the direction of the head can reveal a character's thoughts. FIGURE 4.64	-
62	Eyes Turn Right	Same as 61 but opposite. FIGURE 4.65	-
63	Eyes Up	Eyeballs are rotated upwards in their sockets. Usually, this causes the bottom lids to rise about half way up the eyeball. The simple act of looking up can be used for countless expressions. (people often look up when they pray, as though they are talking to someone above them) FIGURE 4.66	-
64	Eyes Down	When the eyeballs are rotated downwards, this can emphasize the feeling of disdain, feeling of superiority or (depending on the direction of the head) a shameful expression. FIGURE 4.67	-

Figure 4.62.

Figure 4.63.

Figure 4.64.

Figure 4.65.

Figure 4.66.

Figure 4.67.

■ Interpreting Action Units

We now have a pretty comprehensive collection of available actions for the human face. Using this table, we can begin to extract individual blendshapes and joints that will represent the complete list of attributes for our shape set. The task of interpreting the action units for your character is done to simplify the facial setup and weed out extraneous units that might detract from the usability of the setup.

You may have noticed that some of the action units are very similar in movement. These are better thought of as *different intensities of the same movement*. For this reason, not all of the action units need to have their own blendshapes. Remember that some action units can be better simulated using joints instead of blend shapes. Take, for example, AU-25, AU-26 and AU-27. These three action units are, for all intents and purposes, varying degrees of opening a jaw. Knowing this, we can take care of all three of them with one joint, the jaw joint. By combining AU-25 - 27, we have made the setup process faster, and the rig more efficient without sacrificing any flexibility.

Other groups of action units can be combined as well. Let's consider AU-18 and AU-22. AU-18 creates a squished up, puckered lip region, ready for kissing. AU-22 is exactly the same except that the lips are opened slightly allowing sounds to escape the mouth (specifically the 'o' phenome). Rather than sculpting two different blend shapes for AU-18 and 22, we can simply describe AU-22 as a combination of AU-18, AU-16 and AU-10 (AU-18 + AU-16 + AU-10 = AU-22). We can do this because AU-16 and 10 simply pull the bottom and top lips open. You will find that varying intensities of the combined action units will produce very good results.

How you choose to combine and interpret the action units is up to you. More advanced setups may combine very few action units while simpler setups can benefit by squishing as many of them as possible into fewer shapes and joints. The shape set presented in this book is, in my opinion, a very good balance between usability, diversity and ease of use. That being said, you should always remember that individual characters may require a special interpretation of the base action units. This is especially true for fantastical or non-humanoid creatures.

In addition to combining several different action units, some of them must be split up into corresponding left and right versions. Action units do not describe the asymmetrical nature of many of the muscles in the face. All major areas of the face are capable of deforming asymmetrically. For this reason, many of the action units must be interpreted as the combination of their left and right counterparts. For example, AU-9 (the nose wrinkler) is responsible for our ability to sneer. The human face, however, has the ability to sneer asymmetrically. We can wrinkle both the left and right sides of our noses separately. Hence, AU-9 must be split into AU-9 left and AU-9 right.

For simple characters, in a quick production, asymmetry may be sacrificed to save time. A completely symmetrical facial rig uses far fewer shapes and is faster to build. Most productions could use symmetrical facial rigs for secondary or background characters without any trouble.

■ Secondary Actions

Before I dive into the finished shape set, I want to remind you that while Dr.Ekman's facial action coding system accurately and completely describe the capabilities of the human face, it does not cover all of the secondary actions in the face. Secondary actions are important for digital animators (more so than psychologists). The secondary actions include creasing, bulging, stretching and wrinkles. None of these are free, and all of them must be taken into consideration for every shape. One of the most fundamental problems with digital deformations is the loss of volume. Every secondary action is a side effect of the skin, flesh and fatty tissue in the face trying to maintain volume while undergoing deformations. For example, consider a smile that has no creasing in the cheeks. I'm sure you can see how this would look very strange. Adding these details wherever you can will greatly enhance the believability of your creature's face.

The great thing about secondary actions is that they are far less strict, much more chaotic and as individually unique as snowflakes. The wrinkles on my forehead are as specific to me as my finger prints. When sculpting secondary actions into your blendshapes, always have a mirror present (for realistic reference). While sculpting, you might wonder what happens to the corners of your eyes when you squint. With a mirror, you can quickly observe exactly how your crowsfeet wrinkle patterns look, and you can use this to add another level of believability to the shape.

Sculpting this kind of detail into your base mesh requires heaps of edge loops. Chances are that your creature's mesh is *not* dense enough to hold the finest wrinkles and details. While the vast majority of productions do not need this kind of extreme detail (and never in a video game), they can still be added through the use of custom bump or displacement maps. Simple shader networks, with driven bump intensities, can be setup to add minute wrinkle detail at render time. The specifics of setting up said shader networks can easily be figured out with the help of a junior rendering TD. It's not difficult to do, but it does take time to paint all that detail into several custom maps. Try to get as much wrinkle detail into the base mesh first, if more detail is needed, driven shaders are your best bet.

■ The Final Shape Set

After heaps of testing, tinkering, tweaking and re-testing, you should arrive at a final list of features that need to be built into your facial rig. The character we used in this chapter is not entirely human, but his shape set could just as easily be used on a human character. In fact, this shape set can be used, with very little change, for just about any creature that has a human-esque face. Animals are fundamentally the same .

Our character mesh was modeled by a talented modeler by the name of, Kürsad Karatas. The creature is named Zuckafa, and judging by his face, he is probably not a very nice guy. Regardless, we are going to build a shape set that will enable Zuckafa to express himself beyond that cold blank stare.

Zuckafa's base mesh is comprised of 6624 quad faces. His edge flow is pretty good, and will enable us to sculpt some great expression into him. I would consider Zuckafa to be a medium resolution character. Next generation video games will use meshes like his, but for todays games, he is probably a little too dense. If this character were intended for a film or television production, his mesh would need more edge loops in order to sculpt even finer details. For our demonstration purposes, Zuckafa's mesh is a good compromise between being low and high resolution.

Rigging a shape set like this one can take several days. When completed, it must be hooked up to some sort of interface to give the animator an easy way to manipulate the different shapes. The next two sections of this chapter will cover techniques to help you construct the shape set (sculpting blendshapes and placing/weighting joints) and finally, how to bring all the shapes together into a tidy, easy-to-use interface.

Zuckafa's Shape Set:

NAME(S):	TYPE:	DESCRIPTION:
jawJoint	Joint	The jaw is controlled by one single joint. This joint is carefully placed at the jaw's hinge. I prefer to use a joint for opening the jaw (as opposed to a blendshape) for several reasons: 1. Joints deform in arcs, real jaws also rotate on a hinge. 2. The tongue and lower teeth can easily be parented to the jaw. This means that we will not have to create a whole set of extra shapes to keep the teeth and tongue attached to the lower jaw. 3. The weighting on the jaw joint can be easily adjusted, a blendshape cannot. The jaw joint is an extremely important shape and it will likely take some time to perfect the weighting in this area. Detailed coverage of jaw rigging is included in the next section of this chapter. Use: The jaw joint will open and close the mouth (rotate), slide left/right (rotate), and push in/out (translate). Action Unit(s): 25, 26, 27 Area of Influence: This joint has a very large area of effect. The weighting on the jaw should be spread across the entire face and up behind the ears. See exercise 4.1 for a detailed explanation of exactly how to weight a jaw. FIGURE 4.68
neckFatten	Corrective Shape	This shape is not associated with any action units. The neck fatten shape is used to preserve volume in the area under the chin. When using a jaw joint to open the mouth, the region under the chin will not squish and bulge on its own. This shape corrects that. By pulling the vertices under the chin outwards, this shape can make it look like the neck fat is bulging out when the jaw is opened very wide. It may be handy to sculpt this shape in pose space (that is when the jaw is already open). This can be made possible with the use of a script from chapter six. Pose space modeling is discussed in the next section on sculpting shapes. Use: The neckFatten shape is driven by the rotation of the jaw joint. When the jaw is closed, the shape should be off. When the jaw is fully opened, crank this shape up to 100%. Action Unit(s): n/a Area of Influence: The neckFatten shape affects a small region of skin in the corner under the jaw where it meets the neck. The influence should have a nice smooth falloff across the front and sides of the neck. FIGURE 4.69
jawFlex	Corrective Shape	The jaw flex shape does not represent any particular action unit. Simply put, this shape can be used to simulate a clenched jaw. It bulges a couple of vertices at the jaw's hinge. To see what this is simulating, feel the hinge on your jaw as you bite down. Clenching your jaw causes a small protrusion in this area. The shape may only be appreciated by your audience on the subconscious level, but it is so quick to sculpt that this is not really an issue. Use: The jawFlex shape should be set to 0 until the jaw is rotated shut. Set jawFlex to 100% when the jaw is closed past it's rest position. Action Unit(s): n/a Area of Influence: Very small circular region of influence where the jaw hinges to the skull FIGURE 4.70

Figure 4.68.

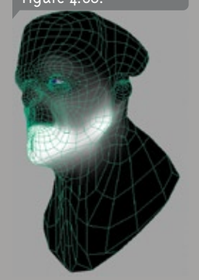


Figure 4.69.

Figure 4.70.

NAME(S):	TYPE:	DESCRIPTION:
neckFlex	Corrective Shape	There are lots of tiny tendons and muscles in the neck that are often completely ignored in animation. Use real life reference when sculpting this shape. The neck should bulge along the length of the tendons as the skin tightens. The neck can be flexed to emphasize extreme anger or when the frown shapes are activated. It is best to animate this shape with a small degree of random noise to make the neck look like it is under tension. Use: The neckFlex shape can be driven by a custom attribute that the animator should have full control of. You may find it effective to have this shape driven to about 50% when the frown shapes are at 100%. Action Unit(s): n/a Area of Influence: This shape has a small amount of influence across the front and sides of the characters neck. The back of the neck is not affected. FIGURE 4.80
leftSmile rightSmile	Shape	The smile is broken into two, symmetrical shapes. The smile shape is extremely important and will probably take the longest to sculpt of all the shapes. A smile involves pulling the corner of the mouth outwards and back. Do not make the mistake of pulling the mouth upwards too much. For a realistic character, the corner of the mouth needn't be pulled any higher than about the width of the top lip. Creasing is very important here. Ensure that you sculpt a large crease running vertically across the edge of the mouth when sculpting this shape. As with all creasing shapes, real-life reference will help a lot. Some character's may form dimples on their cheeks when smiling, keep this in mind. Use: The smile shapes are used for both emotive expression and lip sync. The smile shapes should pull the mouth into the 'e' phoneme when applied at about 50%. Action Unit(s): 12, 14, 20 Area of Influence: This shape must be sculpted very carefully. It is, in my experience, the most difficult shape to sculpt. This is because it's area of influence must not overlap too much into the upper cheeks or middle of the lips. Because it is asymmetrical, you must also be very careful to ensure that the vertices in the middle of the lips are not affected to much. Keep in mind the fact that both the left and right smile shapes must look correct when added together. The smile shapes should have a nice soft affect across the jaw and over the entire cheek. FIGURE 4.81
leftFrown rightFrown	Shape	Like the smiles, the frown shapes are tricky. The corner of the mouth must be stretched wide and pulled downwards slightly. As with the smile, a large crease is formed in a moon shape under the corner of the mouth. Use a mirror and think about your character's facial topology when sculpting the creases on this shape. Frown creases vary greatly from person to person and they rely on several factors including, skin elasticity, bone structure and fatty tissues. As with the smile, you must be careful that the frown is not pulled down too far. If your frown looks really cheesy, it is probably because the vertices have been pulled downwards too much. Use: The frown shapes should be combined into a single control with the smile shapes. They are a very fundamental shape and are used in almost every negative emotion. Action Unit(s): 15 Area of Influence: The frown should influence the entire corner of the mouth and subtly upwards across the chin and upper lip. You may wish to tug a little bit on the tip of the nose too. FIGURE 4.82

Figure 4.80.

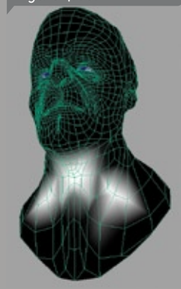


Figure 4.81.

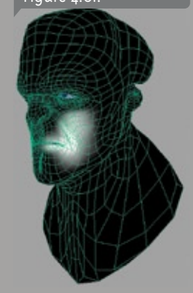
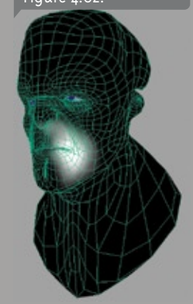


Figure 4.82.



NAME(S):	TYPE:	DESCRIPTION:
pucker	Shape	The most difficult shapes are the ones that require vertices to squish together. The pucker shape is all about squishing the edge loops in the lip to make it look like the character is kissing. To simulate the maintenance of volume, pull the lip vertices outwards and up. Fattening and pulling the lips out will make for a much more appealing pucker shape. If there are enough polygons, add some small creases running vertically over the lips. Be careful to ensure that this shape is compatible with the opened jaw joint. Use: The pucker shape is essential for creating the 'o' phoneme. In addition to lip sync, pucker can be used to augment several different emotions. The pucker shape is very important, take the time to get it right. Action Unit(s): 18, 22, 23 Area of Influence: The pucker influences the top and bottom lips 100%. Tug the skin in the surrounding areas inwards but do so with a quick falloff. FIGURE 4.83
mouthLeft mouthRight	Shape	These are pretty simple shapes and they do nt properly correspond to any particular action units. The mouthLeft/Right shapes are used to push the entire mouth region side-to-side. Use: Animators will appreciate this feature if they ever want to make it look like a character is talking out of the side of their mouth. Set these shapes up on a custom attribute with a range of -10 to 10. Set three driven keys on this attribute: 1. Attribute = 0, Shape = 0 2. Attribute = -10 , left shape = 1, right shape 0 3. Attribute = 10, left shape = 0, right shape 1 These shapes are a prime example of something that is definitely not necessary, yet might be appreciated by the animators nonetheless. Using custom attributes can help de-clutter an interface by hiding attributes that are rarely used. Action Unit(s): n/a Area of Influence: Both of the lips and their immediate surroundings are affected 100%. As with all shapes that make large movements in the mouth region, the falloff should be soft and extend into the cheeks for maximum effect. The soft, cartilaginous, part of the nose is affected by this movement as well. FIGURE 4.84
leftSnarl rightSnarl	Shape	This is a fun shape to use. The snarl is one of the strongest emotive units in the face. A snarl shape will pull the skin on the side of the nose upwards. This squishing causes the skin to erupt into a mass of wrinkles and bulges. You really have to get some wrinkles in this shape to get it's full effect. The wrinkles should spread across the bridge of the nose and down under the eyes. I always sneer at myself in the mirror when I'm building this shape to observe how the wrinkles are formed. Be careful when you are sneering at yourself in a mirror, others may think you have slipped off your cracker. Use: The snarl shapes will have their own controller to allow the animator to apply them separately or combined. They are absolutely essential, regardless of how simple your setup is. Action Unit(s): 9 Area of Influence: The sneer involves the entire section of skin along the side of the nose. Soften the skin's motion across the nose and down across the top lip. While a snarl is almost always accompanied by raising the corner of the top lip, this should not be built into this shape. Leave the lip alone! Each corner of the upperlip will have a separate shape of it's own. FIGURE 4.85

Figure 4.83.

Figure 4.84.

Figure 4.85.

NAME(S):	TYPE:	DESCRIPTION:
leftUpperLipUp rightUpperLipUp	Shape	The left/right upper lip shapes are tricky. You have to carefully build each left/right shape so that when they are applied together, the entire upper lip is pulled upwards to bear the teeth. The middle of the upper lip is the hardest region to perfect on these shapes. The middle is affected by both the left and right shapes, so it must not transform doubly when both shapes are applied. Ensure that each side of the lip is pulled upwards with a slight taper into the middle of the lip. This way, when both shapes are combined, the middle of the lip will rise as expected. Use: The upper lip can be raised to enhance an expression or a piece of lip sync. If both of the shapes are applied at once, the entire upper lip should rise to reveal the teeth and the bottom of the top gums. Action Unit(s): 10, 11 Area of Influence: The upper lip shapes have a small area of influence. Soften the deformation across the region under the nose and slightly into the upper cheeks. FIGURE 4.86
leftUpperLipDown rightUpperLipDown	Shape	Like the upperLipUp shapes, the down shapes must mix well together. When both are applied, the entire upper lip should be stretched down and curled inwards. At full strength, these shapes should cause the top lip to curl completely inside the mouth. Use: These little shapes can be sculpted in to enhance many different expressions and lip sync scenarios. Action Unit(s): 24, 28 Area of Influence: The corner of the upper lip is fully affected with a soft taper into the middle of the lip. The tapering should be such that when both of the shapes are applied, the middle of the lip is pulled straight down. Add some affect across the area between the nose and the upper lip and into the upper cheeks too. FIGURE 4.87
leftLowerLipUp rightLowerLipUp	Shape	The lower lip shapes are an exact mirror of the upper lip shapes. The lower lip up shapes should act like the upper lip down shapes. The lower lip is pulled and stretched over the lower teeth while being curled inside the mouth. The left/right shapes must add together without messing up the middle of the lip. Use: Pulling the lower lip up is essential to many expressions. When lip syncing, the lower lip is heavily involved, especially for the 'f' phoneme. Action Unit(s): 17, 24, 28 Area of Influence: The lower lip up shapes affect the lower lip 100%. Soften the deformation away from the corners of the mouth into the middle of the cheeks. Also important in this shape is pulling some skin over the chin and upwards. There is a muscle in the chin that is activated here and this causes the chin region to stretch. FIGURE 4.88

Figure 4.86.

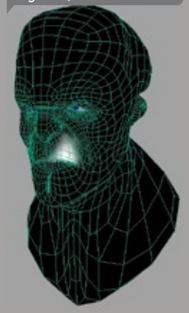


Figure 4.87.

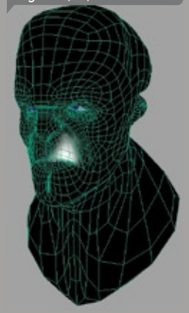
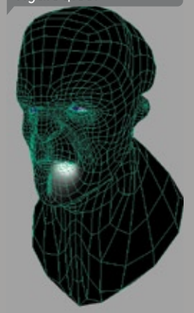


Figure 4.88.



NAME(S):	TYPE:	DESCRIPTION:
leftLowerLipDown rightLowerLipDown	Shape	Not unlike the upper lip up shape, these pull the lip away to reveal the teeth and gums. In addition to being pulled downwards, it is extremely important that the lower lip is pulled outwards slightly as well. Use: Lowering the lower lip can enhance lots of facial expressions. When the lip is lowered enough to reveal the bottom teeth, it can look quite menacing. Action Unit(s): 16 Area of Influence: The corners of the bottom lip are fully affected. Taper the affect into the middle of the lip to ensure that, when mixed, the bottom lip deforms correctly. Add some subtle pulling across the chin and middle of the cheeks. FIGURE 4.89
leftCheekSuck rightCheekSuck	Shape	This shape causes the left cheek to be sucked inwards. The loosest skin is in the middle of the cheeks and should be affected most. Pull the skin inwards just to the point where it collides with the teeth. While this may seem too far, the cheeks will no longer collide when the jaw is opened. To see how the different areas of the mouth and face are affected, look in a mirror while you suck your cheeks inwards. The area under the nose and below the bottom lip are also affected. Use: Sucking the cheeks inwards can be used for many different things. Imagine a character taking a large bite of a sour candy. The 'sour' face involves sucked-in cheeks. Action Unit(s): 13 Area of Influence: Be very careful with these shapes and exactly where they influence. They must mix well together. Be mindful of where the jaw, teeth, and cheekbones are located before sculpting these. Do not break the cheek bone by sucking the skin inwards in this region. The cheeks should suck inwards and wrap around the underlying skeleton. Always remember to add a nice, soft gradation. FIGURE 4.90
leftCheekBlow rightCheekBlow	Shape	This is the exact opposite of the cheeks suck-in shape. In fact, in many cases (like with Zuckafa) you may be able to simply overload the cheek suck shapes to get these. The cheeks need to be pushed outwards, and pulled tight. The area above the upper lip should push out slightly as well. Use: When used very subtly, blowing the cheeks can make the mouth look more organic during lip sync. It can also help some specific expressions. Action Unit(s): 13 Area of Influence: The area of influence here is the same as the cheek suck shapes. FIGURE 4.91

Figure 4.89.

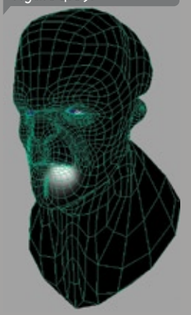


Figure 4.90.

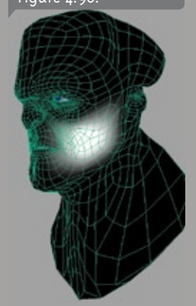
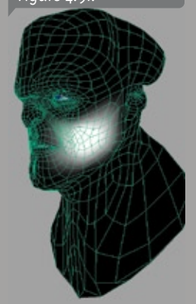


Figure 4.91.



NAME(S):	TYPE:	DESCRIPTION:
leftEyeJoint rightEyeJoint	Joint	These joints are placed precisely in the center of the eyeball geometry. The eyeball geometry can then be parent constrained to these joints. No weighting (or very little) is applied to these joints. Use: These joints act as drivers for the eyeballs. The animator can move a controller curve, which in turn drives the rotation of these joints to make the character look up/down and side-to-side. Rigging eyes with this method is covered later in this chapter in exercise 4.3. Action Unit(s): 61, 62, 63, 64 Area of Influence: The eye geometry is simply parent constrained to this joint. FIGURE 4.92
leftEyeTopLidJoint rightEyeTopLidJoint	Joint	These joints must be carefully weighted to the top eyelids to allow it to slide over the eyeball realistically. In some setups, you may be able to use a blendshape to move the top eyelid. I find that joints are especially well suited to this task because they deform in an arc. The arcing motion can help prevent the eyelid from crashing through the eyeball. This is especially important if the character's eyeballs are large and bulbous. For many characters, blendshapes are simply too linear to deform the eyelids. Use: The top lid joints are used to move the top eyelids. This is essential for many expressions, as well as blinking. Action Unit(s): 5, 41, 42, 43, 45, 46 Area of Influence: The top eyelid joints are very softly weighted across the top of the eyeball. Ensure that the weighting is gradated into the corners of the eyelids. Do not be afraid to add slight amounts of influence up into the brows. This can liven up the face. FIGURE 4.93
leftEyeBottomLidJoint rightEyeBottomLidJoint	Joint	The bottom lid joints are setup exactly like the top lids joints. Having a joint for each eyelid (top and bottom) will enable the animators to pose the eyes exactly as they wish. Use: The top lid joints are used to move the bottom eyelids. This is essential for many expressions, as well as blinking. The bottom eyelids can be very expressive, especially on a cartoon character. I have seen many feature film animated characters that utilize the bottom eyelid to great effect. Action Unit(s): 41, 42, 43, 45, 46 Area of Influence: Like the top eyelid joints, these are weighted to the bottom eyelid geometry. A very soft gradation is spread across the lower lid and under the eye. FIGURE 4.94

Figure 4.92.

Figure 4.93.

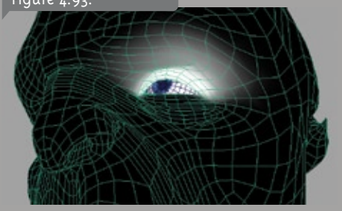
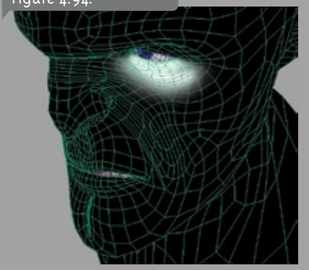


Figure 4.94.



NAME(S):	TYPE:	DESCRIPTION:
leftEyeTugLeft rightEyeTugLeft leftEyeTugRight rightEyeTugRight	Corrective Shape	As part of the 'soft eyes' setup, these shapes tug very lightly on the skin surrounding the eyeball to pull it to the left/right. This is a very subtle shape. When sculpting these shapes, take into consideration the size of the character's eyes. Because Zuckafa's eyes are very deep and quite small, this effect is very subtle for him. Other characters, with larger protruding eyeballs, will have much more movement in this area. Use: These shapes are never directly controlled by the animator. The tug left and right shapes are driven by the rotation of the eye joints. When the eyeballs look to the right, the skin is 'tugged' to the right. When they rotate to look left, the skin is tugged to the left. This setup is covered in detail in exercise4.2. Action Unit(s): n/a Area of Influence: These shapes affect the entire eyeball region. Be sure to spread the influence up into the brow and across the temple in a very subtle manner. FIGURE 4.95
leftSquint rightSquint	Shape	These shapes can take a while to sculpt. The skin under the eyes (the upper cheek) must squish upwards. Skin from the temples is pulled inwards and the famous 'crows feet' wrinkles will appear. Real reference will help you place the wrinkles and determine how much to push the upper cheeks. Use: The squint shapes are hooked up to their own controller. The animator is then able to squint the left/right sides independently, as needed for the current expression. Action Unit(s): 6, 7, 44 Area of Influence: The squint shapes affect the entire region around the eyes. Be sure to pull the skin across the side of the cheeks very slightly. Gradate the falloff into the lower cheeks and across the temples. FIGURE 4.96
innerBrowUp	Shape	The inner brows include the region between eyes. This area can move independently of the outer brow regions. The brow up shape pulls the inner brow upwards. For a realistic character, this area should not be pulled up farther than about half-an-eyeball. Do not squish the brows inwards, this action will be controlled with a separate shape. There is a lot of creasing involved in this shape. Horizontal wrinkles will form patterns across the forehead. As with all creasing, a mirror can say it best. Use: This shape is absolutely essential in order to convey the basic emotion of being sad. Action Unit(s): 1 Area of Influence: The innerBrowUp shape should push up into the forehead. Do not affect the outer regions of the brows with too much intensity. FIGURE 4.97

Figure 4.95.

Figure 4.96.

Figure 4.97.

NAME(S):	TYPE:	DESCRIPTION:
innerBrowDown	Shape	The inner brow down shape pushes the inner brow downwards. This is pretty much the opposite of innerBrowUp. In fact, overloading the innerBrowUp shape can help you get a good start on this shape. Use: This shape is essential for emotive expressions. Pushing the inner brow downwards can quickly create an angry look. Action Unit(s): 4 Area of Influence: Exactly the same as innerBrowUp. FIGURE 4.98
squeezeBrow	Shape	The brow squeeze causes the inner brows to compress together. Squeezing the brows together creates a lot of wrinkles. Some of these wrinkles are definitely large enough to be modeled into the base mesh. Use: This shape is used to get extra mileage out of the innerBrow shapes. When combined with innerBrowUp or innerBrowDown, it creates more intensity. Used on it's own, it can make a character look concerned or concentrated. Action Unit(s): 4 Area of Influence: The inner brow region is completely affected. Pull the skin in a subtle way, across the entire brow. FIGURE 4.99
leftOuterBrowUp rightOuterBrowUp	Shape	The left/right outer brows have their own shapes. Pulling the outer brows up will result in a semi circular brow shape. In a realistic character, the outer brows can be pulled upwards about one eyeball high. Like the inner brow shapes, these create horizontal wrinkles that propagate across the forehead. Use: The outer brows can be posed to create all sorts of expressions. They are essential parts of character animation. Action Unit(s): 2 Area of Influence: The outermost part of the brow is affected the most. Taper this affect into the middle of the brows and across the outside of the eye. Pulling the outer brow upwards should pull skin away from the eye. FIGURE 4.100
leftOuterBrowDown rightOuterBrowDown	Shape	The outer brows are pushed downwards. This is pretty close to the exact opposite of the outerBrowUp shapes. Try overloading the up shapes to get a good starting point for sculpting the outerBrowDown shapes. The brow should be lowered enough to slightly hang over the eyeball region. Use: Like all brow shapes, these are used to convey a particular emotion, or a change in emotion. Action Unit(s): 2 Area of Influence: Same as outerBrowUp. FIGURE 4.101

Figure 4.98.

Figure 4.99.

Figure 4.100.

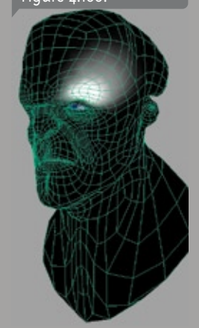
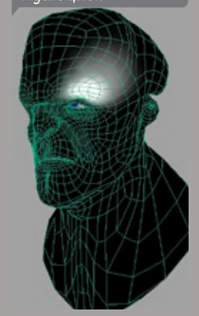


Figure 4.101.



NAME(S):	TYPE:	DESCRIPTION:
head	Joint	This joint is placed at the base of the skull where the spine meets the head. The head joint has the ability to rotate on all three axes. This simulates the last few upper cervical vertebrae. Use: The head joint can rotate the entire skull to get head tilts. The orientation of a character's head can reveal a lot about it's thoughts and feelings. Action Unit(s): 51, 52, 53, 54, 55, 56 Area of Influence: Weighting the head joint is not difficult. Just ensure that it encompasses the entire skull. Gradate the falloff from the head joint into the jaw and neck joints. FIGURE 4.102
neck	Joint	This joint is placed below the head joint and roughly resembles the motion of the lower cervical vertebrae. Like the head joint, the neck can rotate on all axes. Use: The neck joint can help to smoothly pose the head into a more extreme pose. It is also great for creating the head back and head forward action units, something the head joint cannot do on it's own. Action Unit(s): 57, 58 Area of Influence: The neck joint can influence down into the clavicles. Ensure that it has a very smooth falloff going up into the head and down into the rest of the body. FIGURE 4.103
tongue1 - tongueX	Joints	Dr. Ekman's action units do not encompass the tongue. Tongues are a bit of a hassle because I have yet to come across any sort of 'standard' way that tongues are best rigged. That being said, I usually choose to use a simple FK joint chain to control my tongue. Here is why: 1. Blendshapes are way too linear for anything other than the most basic of tongue movements. An FK chain can make the tongue look much more organic as it curls and flops around. 2. Using IK (spline IK is somewhat popular for tongues) on the tongue is overkill. It can be done, but unless the tongue is supposed to be doing something really bizarre, an FK chain will more than suffice. IK can be tricky to setup so that the tongue will remain oriented properly within the mouth. Avoid complexity wherever possible. 3. FK is easy to setup and stable. A good FK tongue rig can be done in 10 minutes. See Zuckafa's finished rig for an example of a tongue done in this way. Use: The tongue is used for accentuating the 'l' phoneme. Monster's always seem to like to lick things too, hence the need for a tongue. Action Unit(s): n/a Area of Influence: The entire tongue geometry. FIGURE 4.104

Figure 4.102.

Figure 4.103.

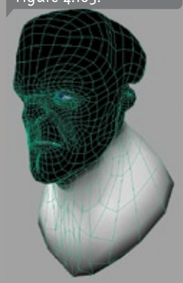
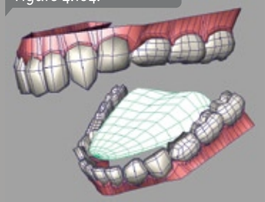


Figure 4.104.



■ A Final Word On Shape Sets

The shape set outlined in this book was designed for the Zuckafa character and will not be the best for *all* creatures. That being said, I believe that this shape set is a good starting place with which you can build your own sets.

Many productions will have several characters with very similar facial topology (same race, age etc...). In these cases, it is best if a standardized shape set is outlined for the entire production. This will help both the animators and the riggers quickly become familiar with all of the caveats of each character's face.

A standardized shape set can be especially useful in a video game production where animation clips can be copied between multiple, different characters. This is already being employed in some modern video games. The insanely popular Half-Life franchise (created by the very talented Valve Software team) has always striven to achieve a higher level of believability. Half-Life 2's characters all use a standardized shape set that is hooked up to a high level animation control dictated by predefined clips and AI technology.

This is just one example of how a shape-set can be varied to produce a production standard that streamlines the facial animation pipeline. Dr. Ekman's FACS was used extensively in the creation of most of the major, modern facial animation masterpieces. Gollum, Half-Life 2 and Zuckafa all have their foundation in this system. Learn it well, then learn how to adapt it to your own productions.

Having a well defined shape set is the first task in creating believable facial animation. Actually constructing the set of blendshapes and rigging the necessary joints is the next step. The following section will provide you with useful tips and techniques for rigging the face. After that, the next section, 'Interfacing With the Animator', will describe how to hook up all the elements to bring this together into a tidy package for the animator to use.

Constructing a Shape Set:

Having made it through the gruelingly meticulous process of creating a shape-set, we are finally ready to start building stuff! Yay!

This section includes a lot of helpful sculpting tricks and techniques. I do not pretend to be a modeling guru (I am definitely not a modeling guru) but I have developed some good workflows for sculpting blendshapes. These techniques will help save enormous amounts of time. Using some freely available MEL tools (from highend3d.com) this section will help you maintain consistencies across lines of symmetry and generally make your shapes look better.

Constructing a shape set involves a couple of steps that, if followed correctly, can turn a potentially disastrous waste of time into a pleasant day of sculpting. In addition to sculpting shapes, we have to take care of joint placement in the face and how to weight joints to the mesh.

■ Preparation and Organization

Before you start pulling vertices, make sure your mesh is *finalized*. Making changes to the character's mesh (this includes UVs) can cause your blendshapes to not function correctly. Changes to the UV's may cause really nasty problems and loads of memory sucking history. Do not start work on an unfinished mesh!

Here is a list of things you can do to prep for constructing a facial shape set:

1. Make sure your mesh is final. (If I sound like a broken record it is because this is **such** an avoidable pitfall)
2. Export the mesh as an .obj and scale it appropriately as described in the 'Model Preparation' section earlier in this chapter.
3. Create a new directory structure to hold all the blendshapes.
4. Duplicate the base mesh (without history) and put this in it's own layer. Name this 'baseShape'. This mesh will not be used except to duplicate in order to start modeling another shape. Keep this mesh in a locked layer and do not touch it except to duplicate it.
5. As you work, save your files in steps. I usually use a lettering convention (smileA.mb, frownB.mb etc...) but the important thing is that you can go back in time if you make a huge mistake (and you **WILL**).
6. Ensure that your modeling environment is setup the way you want it. I use a custom Maya shelf to house my most used tools during the modeling process.
7. Start modeling!

■ Push and Pull Like Clay

For the majority of the blendshapes, you can simply duplicate the base shape and start pulling vertices. Using the move, rotate and scale tool provides all of the control necessary to sculpt any blendshape. What these tools do not offer, is efficiency. Wouldn't it be nice if we could just push and pull on the mesh and have the surrounding vertices behave like clay? Thanks to the new Soft Modification tool in Maya 6.0, this is now possible.

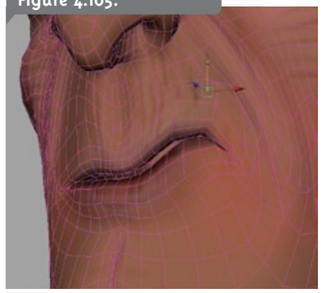
Soft modification is not hard to visualize since it more closely resembles the way physical materials behave. By pulling on a single point on a mesh, the soft mod tool can affect surrounding vertices by a percentage of the total transformation. This falloff is fully adjustable, allowing the artist to control exactly how each push and pull will affect the surrounding vertices.

Soft modification has been around for ages. Competing software packages have had a variation of soft modification for many years. Alias has finally recognized the usefulness of such a tool and so they have included the Soft Modification tool/deformer in Maya 6.0. Better late than never I say.

There are some important things to know when using Maya's Soft Mod tool, especially if you are coming from another package. Every artist (using every software package) has at one point or another experienced feature envy. It happens when you watch a friend or co-worker do something in another package that you cannot do in your own. When it comes to soft modification, I am afraid Maya is somewhat behind the times (it makes up for this in other areas). The problem lies in how Maya's Soft Mod tool determines falloff. Remember, falloff determines the amount of influence surrounding vertices will inherit from your push/pull. Maya calculates this falloff by measuring the distance (in a straight line) from the soft mod point to each vertex. This creates the problem where a soft modification may affect areas of the mesh that you never intended FIGURE 4.105. The problem rears it's ugly head in several areas of the face. Consider the upper lip up shapes. If you create a soft modification on the corner of the upper lip, the vertices on the

lower lip will likely be affected (even with a very sharp falloff). 3ds Max has a soft modification tool with the ability to have it's falloff adhere to the distance *along the surface* as opposed to in a straight line. This may not sound like much, but boy would it be useful for blendshapes. I guess you could say I have a little bit of feature envy.

Figure 4.105.



Regardless, Maya's soft modification tool is still a tremendous help with sculpting shapes. This brings me to the main point I want to make about sculpting in Maya:

Use whatever you can!

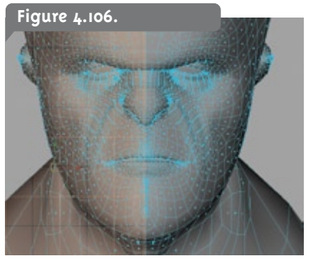
This ambiguous little exclamation means that you should resist getting tunnel vision while sculpting. Do not forget that one of the main reasons that blendshapes are so cool is because you can create them however you want. When sculpting shapes, use a combination of point pushing, soft mod, lattices, joint weighting, clusters, the sculpt polygons artisan brush and even sculpt deformers. Get familiar with everything under the `Animation > Deform` menu and keep an open mind. Do not worry about adding gobs of history because in the end, you can delete the history and 'bake' it all into a shape. All that matters is the end result, the means to get there are irrelevant.

■ Dealing With Symmetry

If you have taken any classical animation, posing, or composition courses, you will immediately associate 'symmetry' with being *wrong*. Indeed, having a perfectly symmetrical character can lead to boring, lackluster performances. The problem, for character riggers, is that asymmetrical meshes require about twice as much setup work because every blendshape (not only in the face) must be created separately for *each side*.

I would like to stress that, if possible, you should try to keep your creature meshes *perfectly* symmetrical. The key word here is 'perfect'. A perfectly symmetrical mesh will enable the setup artist to simply mirror each blendshape (for the face and body) without problems. A perfectly symmetrical mesh can be created by using the polygon mirror tools during construction. Careful attention must be payed to ensuring that the row of vertices running along the line of symmetry are perfectly planar. Without a planar line of symmetry, mirroring blendshapes is impossible. I paid special attention to ensuring that Zuckafa's mesh was symmetrical FIGURE 4.106 . This careful attention enabled me to sculpt his shape set in half the time!

Figure 4.106.



But if I have to have a perfectly symmetrical mesh, aren't my characters going to look cheap and boring? In the world of 3d animation, there are many different ways to make your character asymmetrical besides altering the base mesh. Often times, having an asymmetrical texture map, clothing or hair style is enough to break the curse of symmetry. Further asymmetry can be added by altering the default pose of the character. For example, a salty old sea captain can be setup with a perfectly symmetrical mesh, but be turned into an asymmetrical character by altering the default facial pose to have him in a lopsided scowl. This provides the best of both worlds. Fast, efficient setup time, with an interesting, asymmetrical character.

Of course, this kind of workflow is not always possible. What if the character has only one eye or is missing a limb? Well, in these cases, I can only recommend that you bite the bullet and create all the shapes for both left/right sides. Who ever said animation was easy?

■ Mirroring a Blendshape

Alright, so now you have a perfectly symmetrical mesh. You duplicated the mesh, sculpted the first shape (maybe a smile) and are ready to mirror it. You duplicate the smile blendshape geometry and enter a '-1' into the scale X in order to mirror it across the Z axis. Voila! A perfectly mirrored shape, or is it...

Unfortunately, mirroring blendshapes using the negative scaling method will *not* work. When applied as a blendshape deformer, you can witness this problem as you interpolate the shape between 0 and 1. The mesh will squish up and flip across the Z axis as the shape is turned to 1 (on). To understand why this does not work, you need to understand a little something about vertex ordering and the way Maya stores mesh data.

A polygonal mesh, at it's most basic, is an array of points. These 3d points represent the positions of the vertices in the mesh. If you are familiar with MEL, you could represent this as an array of vectors (if you do not understand arrays and vectors, please revisit the MEL sections in the previous chapters and check the MEL help documentation. This is actually the model for a point cloud, (something similar, but quite different from a polygonal mesh) but because blendshapes have no regard for edges, (they only care about vertex positions) it can help to think of them as a point cloud.

So if we can represent a polygonal mesh as an array of vectors, then that must mean that each vertex corresponds to an index number in the array. Imagine a polygonal sphere named ball. We could represent it's vertices in MEL like this:

```
vector $ball[];
```

Similarly, we can refer to any vertex in this mesh by the index of the array. The tenth vertex in the mesh could be represented by (remember, arrays are 0 based):

```
$ball[9];
```

This is the essence of vertex ordering. Simply put:

Every vertex in a mesh has it's own number.

You probably would have known this already if Maya did not do such a good job of hiding it. To see what the exact number is for a specific vertex:

1. In component mode, select a vertex on a polygon mesh.
2. Open the channel box.
3. At the top of the channel box, you should see the name of the shape node of the polygonal object, followed by the little message "Cvs (click to show)". Clicking this will display the number of the currently selected vertex as well as the local space position of said point.

This seemingly useless concept of vertex ordering has some important implications for mirroring blendshapes. With vertex ordering in mind, think of a blendshape target as an array of *displacements*. Basically, a blendshape target is saying 'move vertex X into position XYZ'.

So, for example, let's say you wanted to create a blendshape for a very simple Pinocchio character. We want a shape that can be used to stretch his nose when he is telling a lie. Let us assume that his mesh is really simple and his nose is only one vertex. To create that blendshape we do the following:

1. We duplicate his mesh. This creates an exact copy of the array of vectors that represented the first mesh. Let's call this new array $target[] .
2. Next, we want to sculpt his nose shape. We can do this by selecting the vertex at the tip of his nose (let's call this vertex number 268). To sculpt the shape, we pull the vertex at the tip of his nose forward in the positive Z axis (14 units), we can also pull it up slightly in Y (4 units). We just created a target shape that looks like he is lying (nose is stretched outwards).
3. This means that $target[268] = 0, 4, 14 .
4. If we apply our newly created target shape as a blendshape, it will displace the 268th vertex on the original mesh by the vector (0,4,14).

This illustrates the importance of vertex ordering. Each vertex on a mesh has it's own number and a blendshape works by displacing each vertex according to it's number. Whenever a change is made to a mesh via the polygon tools, *the vertex ordering will change*. This can cause a target shape to affect the wrong vertices. Let's say we merged Pinocchio's hat mesh to his face mesh. This will cause a complete re-ordering of his vertices. The vertex at the tip of his nose will likely not be #268 anymore. Our 'lying' blendshape is rendered useless.

Sometimes, the vertex re-ordering can be subtle enough (as can be the case with some polygon operations) so that some vertices in the target shape remain properly ordered while others are not. In this case, a blendshape may work on half of the vertices while causing random, unpredictable deformations, on other unintended parts of the mesh. Yuck! To avoid this common pitfall, make sure your mesh is finalized before you starting sculpting shapes. Modeling corrections will likely ruin all of your hard work.

With vertex ordering in mind, we can begin to understand how mirroring a shape is accomplished. If you have your mesh modeled in a perfectly symmetrical way, each vertex on the left side, has a corresponding vertex on the right. When using negative scaling, the displacements you sculpted into your target shape's left side are not mirrored to the corresponding vertices on the right. This is why the negative scaling method does not work. To properly mirror a blendshape, we need a tool that can copy displacements across a line of symmetry between the correct, corresponding vertices.

Unfortunately, there are no tools in Maya that can easily mirror a blendshape. Fortunately, we can use MEL to do this. Before I introduce the solution I use for mirroring blendshapes, I want to talk a little bit about the process of finding solutions for problems with Maya.

So far, I have been showing you how to fix your own problems whenever you encounter one of Maya's limitations. I have been doing this by including extensive MEL lessons at the end of each chapter to explain exactly how the MEL fixes work. This chapter is no different, there is a large section at the end of this chapter dedicated to a MEL fix. But for this example, I want you to understand that you may not have the time (or the knowledge) to script a fix for every problem that crops up.

Part of being a good TD, is knowing where to find help.

Before I set off to program a fix for something, I always check with the community to see if such a fix or workaround exists. Building up a network of reliable help is as essential to being successful in this industry as being able to use a mouse.

For this problem of mirroring a blendshape, I was able to find a solution within minutes. In fact, I found several different scripts available online and from those, I tried and tested until I found one that I liked. I finally settled on a script from highend3d.com by the name of abSymMesh v1.1. This little gem is scripted by the talented Brendan Ross. AbSymMesh.mel enables you to perform all sorts of symmetry operations on a polygonal mesh. I used it extensively during the construction of Zuckafa's shape set.

To install abSymMesh:

1. Download abSymMesh.mel from www.highend3d.com.
2. Paste this script file into your Maya scripts directory.
3. Open Maya and execute 'abSymMesh;' from the command line.

This will bring up the script's user interface FIGURE 4.107. I have found this script to be quite easy to use because this interface is so well designed. To get started, you must select your mesh and click the 'Check Symmetry' button. This may take a few moments (depending on your CPU and the size of the mesh). The script will then spit out a message indicating whether or not the mesh is purely symmetrical. If it is, congratulations, you are ready to start sculpting a shape set. If not, you've got to rework the mesh if you want to use the mirroring features of this (or any other) tool.

Once you know your mesh is symmetrical (and thus 'mirrorable'), select the mesh and load it into the script by selecting 'Select Base Geometry'. The script will check the mesh to ensure it's symmetry, and load it's name into the text field. From here, you can perform several different operations on your target shapes like mirroring and flipping. The script works by comparing the currently selected target against the

base mesh that has been loaded. AbSymMesh is a real pleasure to work with and a fine example of how a simple search through the Internet can yield some very helpful, free tools.

Figure 4.107.

I would like to thank Brendan Ross and all other programmers out there who have helped the Maya community so much by releasing their scripts. As you become more proficient with MEL scripting, I hope that, you too, will release some of your scripts into the wild. It is a great way to find new friends and contacts while gaining renown in this tightly knit industry.

Exercise 4.1

Sculpting the Smile Shapes

This exercise will cover, in detail, the process used to sculpt the left smile shape. Once sculpted, the lesson will finish by showing the user how to create a duplicate target shape with a smile mirrored onto the other side the face. This exercise uses the abSymMesh.mel script file. Please install this script from the DVD before continuing.

1. Open exercise4.1_Start.mb . This file contains only Zuckafa's head, mouth and eye geometry FIGURE 4.108. It has been prepared according to the outlines stated in the earlier section on proper mesh preparation. That includes, scaling, ensuring symmetry (with the abSymMesh tool) and exporting as an .obj file. This is a good starting place to begin sculpting shapes.

Figure 4.108.

2. De-reference the layers so that you can select Zuckafa's head geometry. Select `Edit > Duplicate` and open the options box. Reset the duplicate options (do not duplicate with any inputs) and hit 'Apply'.

3. Move the duplicate mesh to the side. Name this mesh, 'baseMesh'. It is from this mesh that we will duplicate to create the various different shapes. Create a new layer and name it 'baseMeshLayer'. Add the baseMesh object into this layer and hide it while you work.

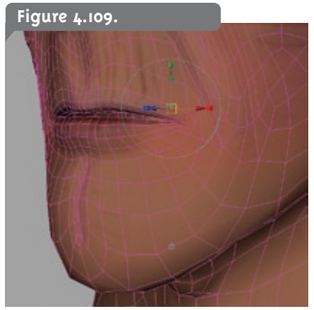
4. You can save this file now. Select the base mesh and duplicate it (again ensuring that no inputs or history are on). Name this shape, 'leftSmile'. It is absolutely important that you name your targets *exactly* as they appear in your list. Without a proper naming scheme, it will be troublesome to connect these shapes with a script.

5. Move this new mesh out of the way and throw it into it's own layer. Hide or template all the other layers to prevent you from accidentally messing up the base mesh while you sculpt.

6. Open exercise4.1_MiddleA.mb to see the file before we start sculpting. Select the leftSmile mesh and press the 'f' key to center the camera on this object. The 'f' key is an essential part of my sculpting workflow as I like to move quickly between shapes without worrying about camera placement.

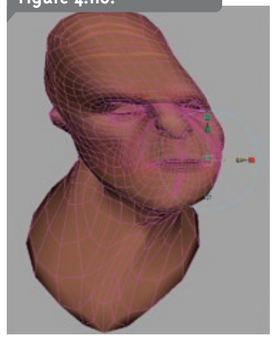
7. Now to start modeling. Before you touch the mesh, remember that a smile must pull the corner of the mouth outwards and very slightly upwards. We also need to create a nice wrinkle where the flesh bulges out. To start things off, let's use a soft modification to pull the corner of the mouth. With the mesh selected, select the soft modification tool from the toolbar. Now zoom-in and click on the corner of the mouth. This will create a soft modification deformer that looks like an 'S' in the viewport FIGURE 4.109.

Figure 4.109.



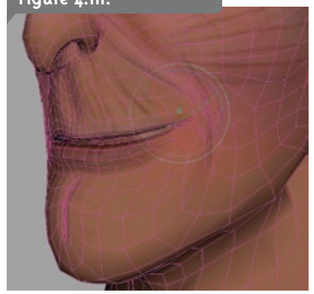
8. Select the soft mod in the viewport and move it around. You will notice that the deformer is affecting vertices in the entire face. The falloff is way to slow FIGURE 4.110. We need to sharpen this up to affect only the mouth and cheek region. With the deformer handle selected, click on 'softMod1' to open the history for the deformer in the channel box. Adjust the falloff radius value to about 0.59. This will give us enough falloff to sculpt some changes without affecting stray vertices.

Figure 4.110.



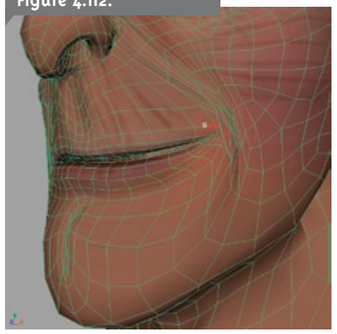
9. Move the deformer handle to stretch the mouth and pull it upwards. You can rotate the deformer handle downwards to lesson the angle of the affected edges. With one swift soft modification, we have the majority of the shape sculpted FIGURE 4.111. Now we need to refine it.

Figure 4.111.



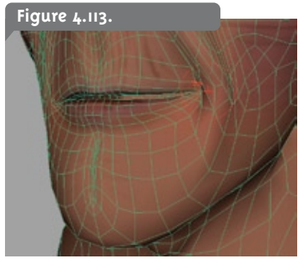
10. Open exercise4.1_MiddleC.mb to see the progress so far. Select the mesh and delete the history, this will get rid of the soft modification handle. Now I am going to introduce a very handy tool for modeling wrinkles, the sculpt polygons tool. Select the mesh and choose `Edit Polygons > Sculpt Polygons Tool`, open the option box. In the tool settings, you can choose from three main operations. You can push, pull or smooth the vertices. This is done through the artisan interface in a very intuitive manner. Since using the soft mod tool on the mouth, you are probably left with a fairly messed up corner of the mouth where the vertices are all bunch up FIGURE 4.112 . The smooth operation can quickly fix this.

Figure 4.112.



11. Brush over the corner of the mouth to smooth the vertices in this area. You may wish to lower the max displacement when using this tool to sculpt smaller changes. With the area smooth out, let's sculpt the crease into the face. Select the 'pull' operation and use the sculpt polygons brush to pull the skin out along the smile crease. Using lots of small brush strokes, you can quickly paint the crease into the mesh FIGURE 4.113. Pull the vertices along the nasolabial fold outwards.

Figure 4.113.



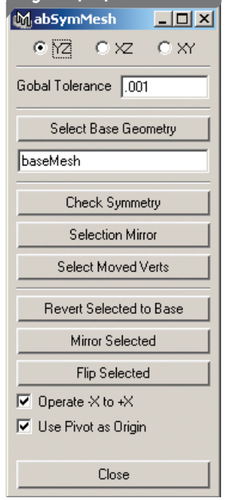
12. Open exercise4.1_MiddleD.mb to see what Zuckafa's left smile looks like at this point. While ninety percent of the shape is done, the last ten percent can take the longest. This is where you get in there and refine the shape by hand. This process involves pushing and pulling, adding subtle skin stretching across the cheek and generally making it look better.

13. It is extremely important to test the shape and see how it looks in action. Select 'leftSmile' and shift select the base mesh. Choose `Deform > Create Blendshape`. I usually test the shape after each major modeling effort. This helps me see what needs to be improved to make the shape look as convincing as possible. Having a mirror on your desk during this phase is essential.

14. When you are finally happy with the finished look of the leftSmile shape, we can automatically mirror it to create the corresponding shape for the rightSmile. As stated earlier, I really like the abSymMesh.mel script for mirroring blendshapes. Load this now by executing ' abSymMesh;' from

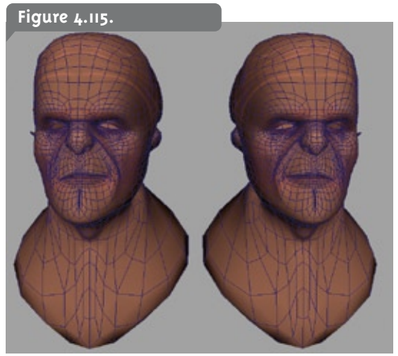
the command line. This will bring up the abSymMesh user interface FIGURE 4.114.

Figure 4.114.



15. Select the base mesh and load it into the script's interface. This should de-gray the rest of the buttons enabling us to create a mirrored shape. Duplicate the leftSmile mesh and rename this 'rightSmile'. With 'rightSmile' selected, click 'Flip Selected'. A progress window will come up as the script does it's calculations. A short moment later, the mesh should appear properly flipped FIGURE 4.115. Tada!

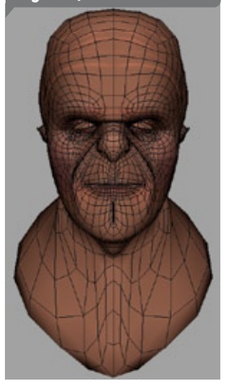
Figure 4.115.



16. To make sure that these shapes are indeed properly mirrored (and not just negative scaled). You can apply them to the base mesh and test it out. The left and right smile shapes should now work both alone and together FIGURE 4.116.

17. If I were sculpting a shape set now, I would hide these shapes in the blendshape layer, then duplicate the base mesh and start all over again on the next shape. Remember to make incremental saves to ensure that you do not overwrite and lose your work.

Figure 4.116.



Smooth Skinning The Face

There are a couple of considerations to be made when you plan to use smooth skinning in conjunction with blendshapes on a facial rig (or any rig for that matter). Firstly, you must have a very good understanding of exactly how smooth skinning works. This will enable you to tailor the weighting in order to produce a very soft, realistic deformation in the jaw, neck and eyelids. Secondly, you must be aware of how the smooth skinned surface will interact with the blendshapes in order to produce predictable and believable results.

Smooth skinning, or smooth 'binding', is the name given to the method that Maya uses to attach a mesh to a joint hierarchy. The technique is pretty straightforward, but there are still a lot of artists that do not fully understand exactly how (and why) it works the way it does. This makes the process of smooth binding a chaotic mess for some people as they struggle to explain why their mesh is deforming in 'strange' ways.

The main culprit here is the infamous *normalization* problem. This problem arises when the setup artist is editing the joint weighting in a particular region of the mesh while inadvertently allowing Maya to screw up the weighting in a different part of the mesh. People who do not understand weight normalization often complain that they are working in circles. Where they fix one weighting issue, another is created. This viscous cycle leads to poor results and lots of stress. I want to explain this problem as clearly as possible in the hopes that it never plagues anyone ever again!

When a mesh is smooth bound to a joint hierarchy, Maya distributes the influence for each vertex across several different joints (as defined by the max influences option). In this way, a single vertex may be affected by the transformation of one or more joints. For example, vertex P may be affected 70% by the elbow joint and 30% by the shoulder joint. The important thing here is that the total influence adds up to 100% (70+30). Vertex P's weighting is said to be 'normalized' to a value of 1.0 or 100%.

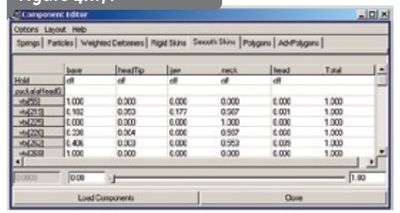
Let's imagine, for a second, that the setup artist has turned weight normalization 'off' (this can be done in the skinCluster node). He then precedes to remove all of the influence from the shoulder joint on vertex P. The only remaining influence on vertex P would then be 30% from the elbow joint. This creates a case where vertex P is not fully affected by the joint hierarchy (it's total influence adds up to only 30%!). If the root of the rig is moved, vertex P will will lag behind by a factor of 70% or 0.7. Vertices with weights that are *not* normalized to a value of 1.0 will lag behind. This is obviously not desirable.

This is where things get interesting. What would have happened had the setup artist left weight normalization 'on' before removing the shoulder influence from vertex P? Well, this would have resulted in the situation where Maya would have first removed the 70% influence from the shoulder (as the setup artist wanted) and then *replaced* it with influence from one or more other joints in the hierarchy (unbeknownst to the setup artist). When you remove or reduce the weighting on a vertex with weight normalization, you are actually telling Maya, "I want you to decide where to put the rest of the influence". Needless to say, Maya does not always do a good job of deciding how to split up the remaining influence. Maya's automatic weight normalization *will* ensure that each vertex has a total weighting of 100%, but it *won't* always add the weight to the right place.

For these reasons, I use a weighting technique that employs *no explicit reducing of weights*. I never (or very rarely) let Maya decide where to add the weighting while I edit smooth skins. To do this, I stick to operations that *add* influence. This way, the only control that Maya has is removing weight from other influences to add to what I want. Never let Maya add influence automatically.

Using a combination of the paint weights tool and the component editor, you will find the weighting process is actually quite easy when you only add weight. The artisan interface is great for fleshing out the larger influences. Minor tweaks can be done in the component editor afterwards FIGURE 4.117. Regardless of what tool you use, try to stick to only adding influence. This will result in a much faster, cleaner, and predictable weighting workflow. The 'scale', and 'replace' operations on the paint weights tool surrender too much control. Do not succumb to the temptation of the normalization monster!

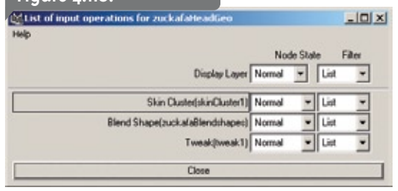
Figure 4.117.



Another common pitfall that beginner setup artists complain about involves *deformation order*. This problem only arises when there are multiple, competing deformers affecting a single mesh. This will always be an issue whenever you combine smooth skinning with blendshapes (as is the case with this facial setup). Maya is very powerful in allowing you to combine different deformers, but it always applies them in the order with which they were created. So if you smooth bound the mesh first, then applied the blendshapes, the blendshapes are at the top of the deformation order. They affect the mesh first. For our facial rig, this is the *wrong* order. We want the blendshapes to be applied *after* the smoothing skinning does it's thing.

It is not hard to tell if the deformation order is incorrect. Basically, the blendshapes will transform the mesh back to it's default pose, even if the head joint is turned to the side. Fixing this is easy. Right-click on the mesh, select `Inputs > All Inputs...` . This will bring up the list of input operations for the mesh FIGURE 4.118. You can middle-mouse drag the items in this list to rearrange the order with which Maya will calculate them. To fix the blendshape issue, simply middle-mouse drag it below the Skin Cluster. It's that simple.

Figure 4.118.



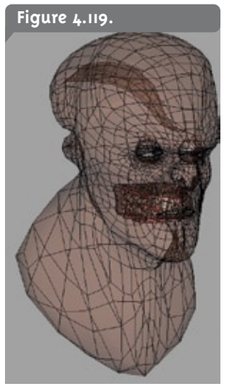
Exercise 4.2 Smooth Skinning the Jaw

The weighting on the jaw is one of the most important parts of a believable facial rig. Weighting a jaw can be a lot of fun. It is really cool to see how the deformations can develop to produce a stretchy, fleshy, organic looking face. Even if you choose to use blendshapes for rigging a jaw to open, it is still very handy to use joint weighting to sculpt the shape.

For this exercise, we will be starting with Zuckafa's geometry. From there, we will place the joints, connect the mouth bit (teeth and tongue) and weight the jaw.

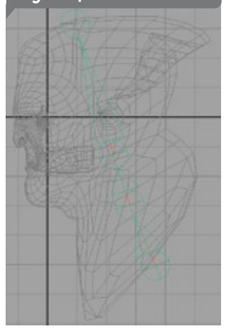
1. Open exercise4.2_Start.mb. This file contains Zuckafa's head geometry. This includes one solid mesh for this skin and mouth bag, plus several pieces for the teeth, gums and tongue FIGURE 4.119. The rigging has not been started yet, so we need to start by placing the head joints.

Figure 4.119.



2. To get a better view while placing joints, drop into the orthographic side viewport. Drawing the joints from this camera will ensure that they are perfectly aligned along the YZ plane where X=0. In other words, the exact middle. Select `Skeleton > Joint Tool` and reset the tool settings. Start by placing a joint at the base of the mesh then create three more joints at the positions indicated in FIGURE 4.120. Press enter to finish the joint chain. Name these, base, neck, head and head tip, in that order.

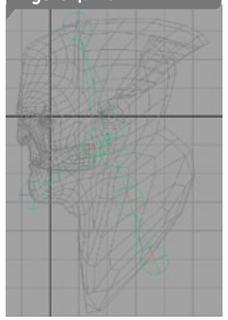
Figure 4.120.



3. The head tip joint is not necessary for weighting. The head tip joint is used as a placeholder so that we can better visualize what direction the head is pointing in. You may choose to not include this joint. This will reduce the amount of influences (always a good thing) but I like having it.

4. Now we need to place the jaw joint. Activate the joint tool again and place two joints like in FIGURE 4.121. The first joint must be placed in the exact position where you want the joint to rotate from. This may require some trial and error. Just think about where the jaw should pivot from and try to place it as close to that as possible. The second joint is the jaw tip joint. Again, this tip joint is not necessary but it does orient the joint in the direction of the jaw while providing a nice indication of how the jaw is rotated.

Figure 4.121.



5. Parent the jaw joint to the head joint.

6. Lastly, we need to place the eye joints. For the soft eyes setup described in exercise 4.3, we will need three joints per eye. The eyeball geometry can be parented to one of these joints. The other two are weighted to the top and bottom eyelids.

7. All of the eye joints must be placed in the *precise* center of the eyeball geometry. If they are simply roughed into place, the eyeballs may rotate off-axis. Even a slight off-axis rotation will look very bizarre. This presents an interesting problem. How can we snap a joint to the precise center of the eyeball geometry? Unfortunately, non of the native snapping tools will suffice. There is no snapping mechanism in Maya that works on the center of an object (only points, curves and the grid). For this problem, I have developed a handy little MEL solution:

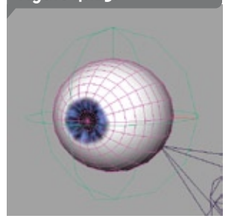
```
proc moveToObjPivot()
{
        string $currentSel[] = `ls -sl`;
        string $base = $currentSel[0];
        string $target = $currentSel[1];

        float $worldSpaceOfBase[] = `xform -query -worldSpace -
                                        rotatePivot $base`;
        setAttr ($target + ".translateX") $worldSpaceOfBase[0];
        setAttr ($target + ".translateY") $worldSpaceOfBase[1];
        setAttr ($target + ".translateZ") $worldSpaceOfBase[2];
}
```

9. Copy the MEL code above (or copy the moveToObjPivot.mel file from the DVD to your Maya scripts directory). When executed, this little chunk of code will enable us to snap any object (like a joint) to the *exact center* of any other object. Snapping to the center of an object is always an issue when rigging spherical things (like eyes or mechanical socket joints). This script does not actually snap to the center of an object, per se, rather it snaps to the object's rotate pivot point. It just so happens that the rotate pivot is usually the center. The script uses the xform command (a very handy command I must say) to query the world position of the snap object's rotate pivot. The pivot coordinates are then used to move the target object. Pretty straightforward, but a good example of how a simple script can add useful functionality.

Figure 4.122.

10. Activate the skeleton tool again and click once in the viewport to create a single joint, then press 'Enter' FIGURE 4.122. Now select the eyeball geometry that you want to snap to, then shift select the newly created joint. Type 'moveToObjPivot' into the command line and hit 'Enter' to call the script. The joint should snap to the eyeball. Their pivots are now perfectly aligned FIGURE 4.123.

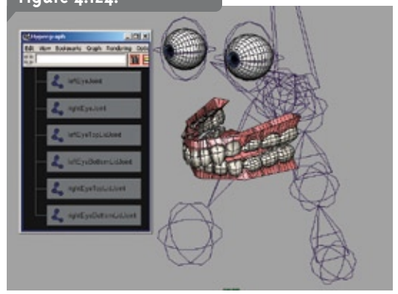
Figure 4.123.



11. Duplicate this eye joint twice. This will leave you with three joints. One for the eyeball, and one for each eyelid (top and bottom). Repeat step 9 and 10 for both the left and right eyes. Name the joints leftEyeJoint, leftEyeTopLidJoint and leftEyeBottomLid joint for the left eye. Name the right eye joints accordingly.

12. When you are done, this should leave you with a total of six eye joints (three for each eye). Parent all of these under the head joint. This concludes the joint placement for Zuckafa FIGURE 4.124.

Figure 4.124.



13. Open exercise4.2_MiddleA.mb to see the finished joint placement. For most rigging endeavors, now would be the time to orient the joints. But because we placed the joints in an orthographic viewport, this is not necessary. The default orientations should be adequate. That being said, it is always a good idea to check and make sure. Go through the hierarchy and rotate each joint to ensure that there are no bizarre orientations that will cause problems after skinning.

14. With the joints all finished, it is time to attach the skin. When binding a mesh to a joint hierarchy, Maya allows you to specify if you want *every* joint to be an influence, or only the currently selected ones. For the soft eyes rig, the leftEyeJoint and rightEyeJoint joints should not be bound. We also do not need to bind the head and jaw tips, if we do not want to. In the hypergraph, select each joint individually, being careful not to select the tip joints or the eye joints (we *do* want the *eyelid* joints, just not the eye joints). With these joints selected, shift select the mesh and choose `Skin > Bind Skin > Smooth Bind`. Open the options and choose 'Selected Joints' instead of 'Complete Skeleton'. Besides this change, the default options will work fine. Click 'Bind Skin' to attach the skin FIGURE 4.125.

Figure 4.125.

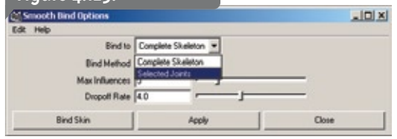
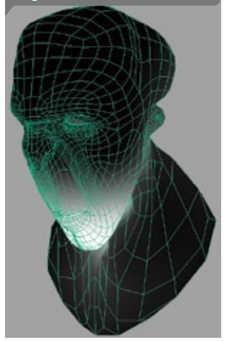


Figure 4.126.

15. Open exercise4.2MiddleB.mb to see the default skinning results. The default weighting is pretty bad. To see how bad, just open the jaw joint. You can see that the jaw is affecting the entire face FIGURE 4.126. To fix this, we are going to completely remove all of the influence from the joint and start from scratch. Open the paint weights tool `(Skin > Edit Smooth Skin > Paint Skin Weights Tool)`. Open the tool options and scroll through the list of influences until you find the joint named 'jaw'. Selecting the influence will show a greyscale representation of the weighting on Zuckafa's mesh FIGURE 4.127. Bright white represents a very strong influence. Darker shades of grey show the falloff.

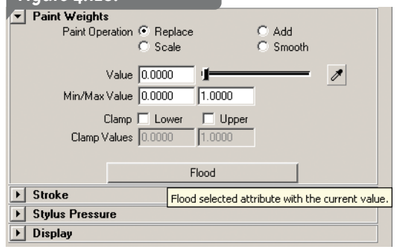
Figure 4.127.



16. With the jaw influence selected, choose the 'replace' operation under the paint weights tab in the paint skin weights tool settings. Set the value slider to '0' and hit the 'Flood' button FIGURE 4.128. This will effectively remove all of the influence on the jaw. Since weight normalization is 'on' by de-

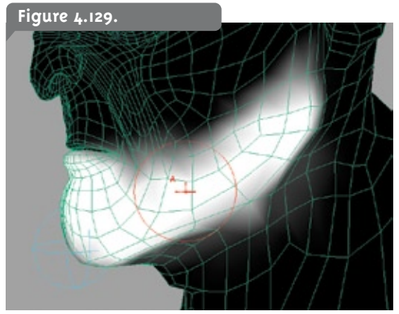
fault, it means that other joints in our list will now be affecting the vertices that were previously affected by the jaw. I know, I said that letting Maya normalize weights for you is a bad thing, but this early in the process, there is no weighting work that Maya could screw up. Flooding weights to 0 is dangerous if you have already done a lot of work.

Figure 4.128.



17. To get the jaw joint to open the mouth, we need to start adding influence back onto the jaw joint. Select the 'Add' paint weights operation and set the 'value' slider to a value of about 0.25. Paint across the left side of the jaw and chin and slightly into the cheeks and neck FIGURE 4.129.

Figure 4.129.



18. To see how this looks, let's first mirror the painting we did on the left side, to the right side. Select the mesh, and choose `Skin > Edit Smooth Skin > Mirror Skin Weights`. In the mirror weights options, choose the YZ axis and check the "Positive to Negative" checkbox FIGURE 4.130. Hitting the mirror button will copy the weighting across the YZ axis. This effectively cuts our weighting work in half!

Figure 4.130.

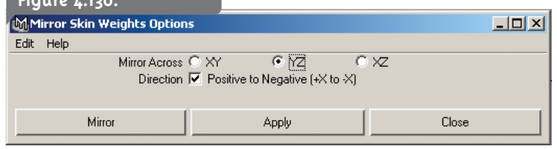
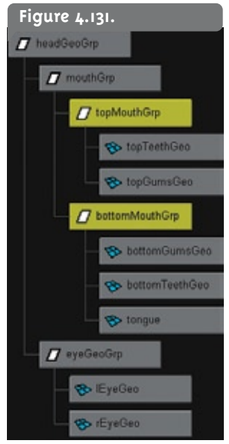
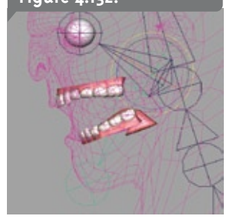


Figure 4.131.



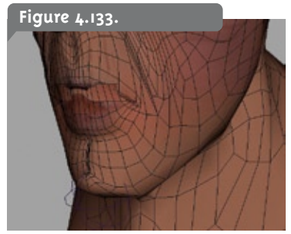
19. Before we continue with weighting the details of the mouth and jaw, we need to attach the teeth and gums. Put Zuckafa's jaw joint into it's default pose. Open the hypergraph and find the group named 'headGeoGrp'. This group node contains all of the geometry for the face. I also organized the teeth, tongue and gum geometries into upper/lower jaw groups FIGURE 4.131. Find the group named 'bottomMouthGrp'. Parent constrain (`Constrain > Parent`, default settings) this group node to the jaw joint. Parent constrain the top mouth group to the head joint. Now the mouth geometry is attached to the skeleton. It helps to have the bottom teeth attached when weighting the mouth region FIGURE 4.132.

Figure 4.132.



20. Select the jaw joint and rotate it to see how the mirrored weighting looks. Chances are that it will not look very good. At this point, you are lucky if the mouth even opens without stray vertices being left behind FIGURE 4.133. This is normal. It can take quite a bit of caressing to get the influence to look correct.

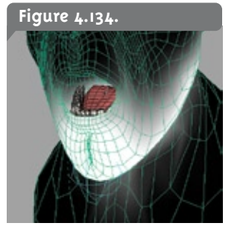
Figure 4.133.



21. It is often handy to have the mesh in a deformed state while painting weights. This is no problem. Just open the jaw slightly and activate the artisan brush again. This time, you will be able to see the effect of the painting as you brush over the mesh. Vertices will snap into place as you add influence. Use a very small (.05) value with the 'Add' operation to brush influence across the face. A convincing jaw joint should spread subtle deformation across the cheeks, eyes and nose. Using a very small value (0.05 or less), will prevent you from making any hard mistakes. Run some very soft influence behind the ears and over the temples. I even like to add a little bit across the brow. Just a tiny bit. While painting, constantly rotate the jaw to test the weighting affects.

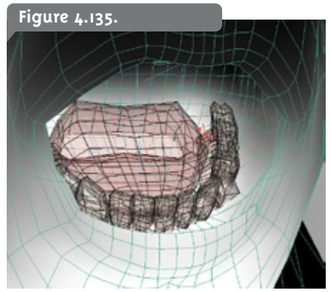
22. Painting weights can be a very back-and-forth task. Constantly paint, test, mirror and re-test. When you get to the point where you are liking how the jaw is opening, you must begin the details. The lip region requires some extra attention. Weight the upper lip so that it stretches over the upper teeth when the jaw is fully open. At this point, it is also necessary to weight the corners of the mouth to make them look like they are stretching. With the jaw fully opened, try to paint the lips into an 'o' shape. Chances are that you will not get a perfect 'o', but getting as close as possible will result in a nice organic looking lip region FIGURE 4.134.

Figure 4.134.



23. Finally, lets take care of the mouth bag. This area is really easy to weight. Just fully open the jaw joint, then add influence to the bottom of the mouth to push these vertices down. Do this until the teeth are no longer hidden by the mouth bag mesh FIGURE 4.135. It may be handy to hide/unhide the teeth geometry while you paint on the inside of the creatures mouth.

Figure 4.135.



24. Do one final test to ensure that you've taken care of any stray vertices. Save the file out and you are done! An experienced weight painter should be able to weight a jaw in less than half and hour. If you find yourself struggling, it may be best to start over with a fresh scene file. Keep trying until you can do it quickly and easily.

Painting weights has such an artistic workflow, that it is often times quite difficult to explain on paper. I highly recommend watching the exercise4.2 video included on the DVD. In this video, you can watch as I go through each step of this process and explain my rationale along the way.

Remember that constructing a shape set, with all of the sculpting and weighting, is the longest phase in the setup pipeline. Do not be surprised if it takes you a week to get it right. With time, you will get faster, but the important thing is that the final product is believable. The next section covers the process of building high-level controls for the animator to use. Remember, an advanced setup is useless if the animator's cannot use it with ease.

Interfacing With The Animator:

Higher Levels of Control

Building an interface for your facial rig is essential. You could build the best rig in the world, but if the animator cannot use it, it is junk. Unlike a shape-set, there are no solid 'rules' to follow when building an interface. There are many different ways to give your animators control. The interface presented for Zuckafa, is a sort of mixture of some of the more popular types.

Usually, an interface, in Maya, takes the form of a NURBS curves. A curve based interface is easy to build, and non-renderable. I have seen interfaces built out of polygon objects too. With a set of curves that represent the various blendshapes and joints in the face, the animator can simply translate said curves to affect the facial rig FIGURE 4.136. Keyframes are then set on these controller objects. In this way, the animator is animating a high-level control that is, in turn, affecting the individual blendshapes and joints in the rig.

Figure 4.136.

Maya provides several different choices for connecting controller objects. Many people prefer the simplicity of a MEL expression. Others may prefer to use set driven keys or even utility nodes. Regardless of what method you use, it should be scripted. Having all of the controller connections collected into one large script file will enable you to quickly

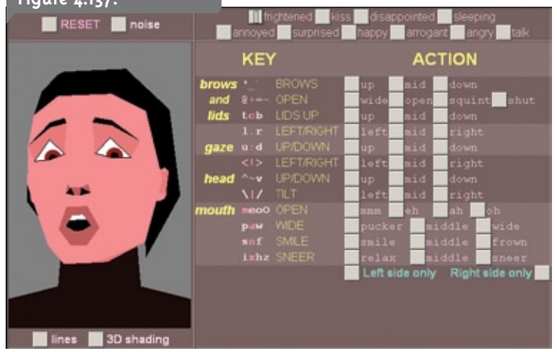
tweak and change the way any shape or joint is affected.

So how does one go about writing a script to connect all of the shapes and joints to controller curves? It is actually quite simple. When I am creating a connection script, I have the script open in my MEL editor window, while I observe it's affects on the facial rig. If a controller is behaving incorrectly, there is no need to completely re-do the driven keys or mess about in the expression editor. I can quickly make the changes in my script file and execute it with the click of a button. As an added advantage, this file may be reusable between multiple characters in a production. This advantage alone should be more than enough reason to consider using a script for simple UI connections.

A discussion on complex high-level facial animation techniques would not be complete without mentioning the work of a brilliant scientist by the name of Dr. Ken Perlin of NYU. Dr. Perlin is perhaps best know for his pioneering work on procedural texturing during the 1980's. Dr. Perlin won a technical achievement Oscar in 1997 for developing the Perlin noise equation. These allow shader artists to generate seamless texture files that simulate natural phenomena using mathematical equations. Recently, Ken Perlin has focused his research into the exciting new area of high-level facial animation. That is, facial animation with automation.

Dr. Perlin has also recognized the importance of assigning high-level control to the muscles that make up the expressive human face. His work is based on Paul Ekman's FACS (just like Zuckafa) and utilized a very similar shape set to the one described here. He has published a great deal of his work online including a really cool Java applet that features an interactive female character. This Java applet features buttons for making the character's face blend into different emotions (ie. frightened, annoyed, sleepy) FIGURE 4.137. It is a lot of fun to play with and can be found here:

Figure 4.137.



http://mrl.nyu.edu/perlin/experiments/facedemo/

For those interested in further reading, Dr. Ken Perlin's 1997 SIGGRAPH technical sketch is available from:

http://mrl.nyu.edu/projects/improv/sig97-sketch/

Dr. Perlin's work represents the next evolutionary step in creating quantifiable, believable facial animation. With a high-level control, animators are freed from the tedium usually involved in posing advanced facial rigs. For the purposes of this book, I want to show you exactly how to build a similar pose library type interface using MEL. Using the pose library system introduced in this book, your animators will be able to simply click a button to put Zuckafa's face into any pose. Poses can then be saved into a special pose file format. These 'pose files' can then be shared between animators and across a network. The MEL lesson at the end of this chapter covers, in detail, how to script such a system.

Before getting into the MEL, I want to show you a little bit more about how higher level interfaces can be built to help your animators.

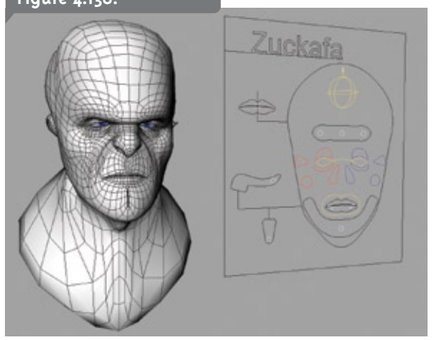
Exercise 4.3 Rigging Soft Eyes

In this exercise, we are going to cover a technique that can be used to rig a character's eyes such that they deform the surrounding skin in a realistic fashion. The entire eye region is very reactionary. That is, it squishes and slides in reaction to the direction the eyeball is looking. In addition to this subtle deformation, the eyelids also move with the eyeball. To see this for yourself, put your finger on your top eyelid and then look around. You should be able to feel your eyeball sliding underneath and tugging on your lid.

This phenomena can be well simulated by setting up some special relationships between the direction of the eyes and the orientation of the eyelid joints. Two blendshapes, one per eye, will also be added into the mix. This is a perfect example of how to apply a high-level control to your rig to help with animation. Sure, the animators could control this kind of behavior explicitly, but why give them the headache when an automated approach can do the same thing?

To set this up, we will be writing a script to connect these relationships to a controller curve.

Figure 4.138.



1. Open exercise4.3_Start.mb. This scene file contains both Zuckafa's final, weighted head rig, plus the curve based interface used to control his facial shapes and joints FIGURE 4.138. Before continuing with this exercise, it might be a good idea to take a look at how the head is rigged. If you look in the hypergraph, you will notice there are a total of six different joints in the eyes (three for each eye) FIGURE 4.139 . Of the three joints per eye, one is weighted to the bottom lid, one to the top lid and the other acts as a parent for the eyeball. Also notice that the mesh has a blendshape node on it. The node is named 'zuckafaBlendshapes'. This blendshape node contains two shapes of particular interest to this setup. They are the leftEyeTugIn and rightEyeTugIn shapes. These shapes are described in the shape set section, earlier in this chapter. To get the left/right eyeTug*Out* shapes, I just overloaded the tug*In* shapes. For Zuckafa, this worked just fine and saved me the hassle of creating two extra shapes.

Figure 4.139.

2. For this setup I decided to drive everything from the leftEyeJoint and rightEyeJoint joints. Use a combination of expressions and set driven keys to create a setup that is both controllable *and* automatic. Before we get into the details, it is important to understand the behavior that we are trying to achieve. Here is a breakdown of what the eye joints affect as they rotate on their Y and X axis (side-to-side and up/down respectively):

1. Looking left: When the eye joints are rotated to the left, the eye lids remain unchanged. For this motion, set some driven keys to drive the tugging blendshapes. When the eyes are looking left, the leftEyeTugIn shape is overloading to a value of '-1'. This causes the skin surrounding the left eye to be tugged outwards, or to the left. The rightEyeTugIn shape is set to a value of positive '1'. This tugs the skin inwards. FIGURE 4.140

Figure 4.140.

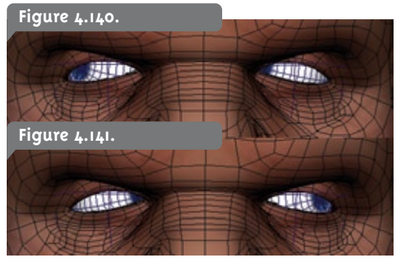


Figure 4.141.

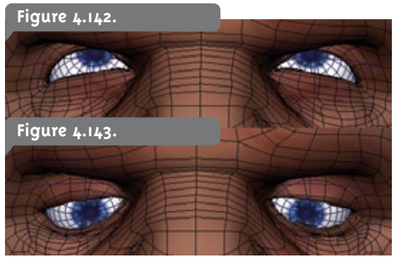
2. Looking right: In this case, the eye joints are rotated in Y to look to the right. Via set driven key, the left and right EyeTugIn shapes are set to the opposite values for looking left. FIGURE 4.141

3. Looking Up: When the eyeball joints are rotated upwards (in Y axis) the bottom eyelid joint should rotate about half way up the eyeball. This actually happens in real life. The top lid should pull up as well. This clears the iris so that you can look up. FIGURE 4.142

4. Looking Down: When the eyeball joints are rotated downwards, the top eyelid should follow and stay in contact with the top of the iris. In this case, the bottom eyelid squishes up a bit to accommodate the iris' new position. FIGURE 4.143

Figure 4.142.

Figure 4.143.



3. To make it even easier for the animator, the eye joints (which are driving the eyelids and the tug shapes) are then driven by one single controller curve. This controller is hooked up via a simple expression to the eye joints. Locate the controller named EyeLookAtControl in the exercise4.3_Start.mb scene file. This controller is a simple NURBS circle that has had limits placed on it's translate X and Y. If you try to translate the curve, it will remain bounded by the limits that were placed on it FIGURE 4.144. It's range of motion is from 1 to -1 in both X and Y. Limits can be placed on any transform channel on any object. To place a limit, open the attribute editor and find the 'Limit Information' tab FIGURE 4.145.

Figure 4.144.

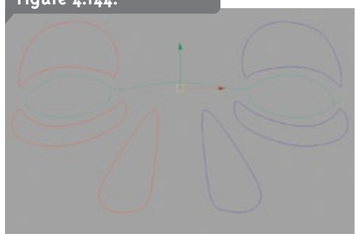
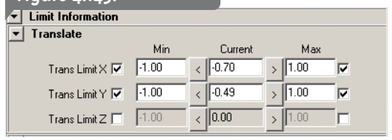


Figure 4.145.



4. Before we start explaining this setup any further, you may wish to see how it looks in action. Seeing this in action is better than any description I can give here. Without further ado, let's open a new text file and start writing the script to make all the connections. To organize the script into one command, (this command can be called from a

button on your shelf) we are going to enclose the whole script into a procedure.

```
proc connectZuckafaFace ()
{
        //Commands go here....
}
```

5. Now, to connect Zuckafa's face, all we need to do is call this procedure from the command line. To make this command a shelf button, type 'connectZuckafaFace' into the command line, then select the text and middle-mouse drag the command into your shelf. This will create a new button. Now you can simply click this button to setup his face.

6. To start with, let's connect the EyeLookAt controller. This way, we can test the setup as we progress. To connect the EyeLookAt controller, we will use an expression. You could just as easily use set driven keys (and there would be nothing wrong with that). For this setup, we are going to be using both driven keys and expressions. This way, you can get a feel for how to script both.

```
leftEyeJoint.rotateX = leftEyeLookAt.translateY*-30;
rightEyeJoint.rotateX = rightEyeLookAt.translateY*-30;
```

These lines of code represent the expressions that will cause the eyeballs to rotate side-to-side. Notice the use of the factor '30' to give the eyes a range of -30 to 30 degrees.

```
leftEyeJoint.rotateY = leftEyeLookAt.translateX*45;
rightEyeJoint.rotateY = rightEyeLookAt.translateX*45;
```

The two line of code above will control the rotation of the eyeballs in the Y or up/down direction. Notice, I chose to give this movement a range of -45 to 45 degrees. These are very simple expressions. Nothing magic going on here. But if we paste these into a script file, as they are, it will not create an expression for us. Like everything in Maya, creating an expression is done through a MEL command. That command is simply 'expression'. The 'expression' MEL command has a couple of interesting flags. With the '-name' flag, we can specify a name for the expression (very useful for debugging). With the '-string' flag, we can specify a string value that represents the actual expression. To rewrite the expressions in MEL, it would look like this:

```
expression -name "EyeLookAtExpression" -string "
leftEyeJoint.rotateX = leftEyeLookAt.translateY*-30;
rightEyeJoint.rotateX = rightEyeLookAt.translateY*-30;
leftEyeJoint.rotateY = leftEyeLookAt.translateX*45;
rightEyeJoint.rotateY = rightEyeLookAt.translateX*45; " ;
```

This MEL command will create an expression for us called 'EyeLookAtExpression'.

7. The EyeLookAtExpression will now control the direction of the eyeballs, and we can tack-on a couple extra lines to make it also control the tugIn shapes. Add these two lines to the EyeLookAtExpression in the connect face script file:

```
zuckafaBlendshapes.leftEyeTugIn = leftEyeLookAt.translateX*-1;
zuckafaBlendshapes.rightEyeTugIn = rightEyeLookAt.translateX;
```

8. That wasn't so bad! If you now source and execute the script, the eyeballs will react to the EyeLookAtControl controller curve. Pretty cool, but we need the eyelids to get in on this action too! To setup the eyelids, I chose to use set driven keys. Do not be afraid to mix driven keys with expressions. Each have their own strengths and weaknesses. Recall that with a driven key, you set the driver value first, then you set the driven values, followed by setting the key. When we are doing driven keys with MEL, it's no different.

```
//Set the initial key, all values set to 0
setAttr "leftEyeJoint.rotateX" 0;
setAttr "leftEyeTopLidJoint.rotateX" 0;
setAttr "leftEyeBottomLidJoint.rotateX" 0;
setDrivenKeyframe -cd leftEyeJoint.rotateX ...
leftEyeTopLidJoint.rotateX leftEyeBottomLidJoint.rotateX;

//Set the key for when the eyeball is rotated up (negative
//rotation in X)
setAttr "leftEyeJoint.rotateX" -30;
setAttr "leftEyeTopLidJoint.rotateX" -20;
setAttr "leftEyeBottomLidJoint.rotateX" -20;
setDrivenKeyframe -cd leftEyeJoint.rotateX ...
leftEyeTopLidJoint.rotateX leftEyeBottomLidJoint.rotateX;

//Set the key for when the eyeball is rotated downwards
//(positive rotation in X)
setAttr "leftEyeJoint.rotateX" 30;
setAttr "leftEyeTopLidJoint.rotateX" 15;
setAttr "leftEyeBottomLidJoint.rotateX" 25;
setDrivenKeyframe -cd leftEyeJoint.rotateX ...
leftEyeTopLidJoint.rotateX leftEyeBottomLidJoint.rotateX;

//Reset the eye joint to 0.
setAttr "leftEyeJoint.rotateX" 0;
```

9. The code in step 8 will setup the eyelids to react to the orientation of the eyeballs. If the code does not make sense, try to recognize that each section (separated by white space) represents one set driven key. Firstly, the 'setAttr' commands set the value of the driver and the driven objects. Then, the 'setDrivenKeyframe' command sets a key for the current values of the specified objects. In total, there are three keys. The first key is set when the eyeball is in it's default pose, the second key is for when the eyeball is looking up and the last one is for looking down. Together, these three keys are all that is needed to automatically drive the eyelids.

10. Copy and paste the left eyelid set driven key commands and replace occurrences of 'left' with 'right'. Because everything is named according to a strict convention, this is all that is needed to setup the right eyes. By using a script, you have effectively cut the setup time in half because you can mirror the setup with a simple copy/paste! The entire connection script should now look like this:

```
proc connectZuckafaFace ()
{
  //Eye look at expression
  expression -name "EyeLookAtExpression" -string "
  leftEyeJoint.rotateX = leftEyeLookAt.translateY*-30;
  rightEyeJoint.rotateX = rightEyeLookAt.translateY*-30;
  leftEyeJoint.rotateY = leftEyeLookAt.translateX*45;
  rightEyeJoint.rotateY = rightEyeLookAt.translateX*45; " ;

  //Left eyelids SDK
  setAttr "leftEyeJoint.rotateX" 0;
  setAttr "leftEyeTopLidJoint.rotateX" 0;
  setAttr "leftEyeBottomLidJoint.rotateX" 0;
  setDrivenKeyframe -cd leftEyeJoint.rotateX ...
       leftEyeTopLidJoint.rotateX leftEyeBottomLidJoint.rotateX;

  setAttr "leftEyeJoint.rotateX" -30;
  setAttr "leftEyeTopLidJoint.rotateX" -20;
  setAttr "leftEyeBottomLidJoint.rotateX" -20;
  setDrivenKeyframe -cd leftEyeJoint.rotateX ...
       leftEyeTopLidJoint.rotateX leftEyeBottomLidJoint.rotateX;

  setAttr "leftEyeJoint.rotateX" 30;
  setAttr "leftEyeTopLidJoint.rotateX" 15;
  setAttr "leftEyeBottomLidJoint.rotateX" 25;
  setDrivenKeyframe -cd leftEyeJoint.rotateX ...
       leftEyeTopLidJoint.rotateX leftEyeBottomLidJoint.rotateX;

  setAttr "leftEyeJoint.rotateX" 0;

  //Right Eyelids SDK
  setAttr "rightEyeJoint.rotateX" 0;
  setAttr "rightEyeTopLidJoint.rotateX" 0;
  setAttr "rightEyeBottomLidJoint.rotateX" 0;
  setDrivenKeyframe -cd rightEyeJoint.rotateX  ...
     rightEyeTopLidJoint.rotateX rightEyeBottomLidJoint.rotateX;

  setAttr "rightEyeJoint.rotateX" -30;
  setAttr "rightEyeTopLidJoint.rotateX" -20;
  setAttr "rightEyeBottomLidJoint.rotateX" -20;
  setDrivenKeyframe -cd rightEyeJoint.rotateX ...
     rightEyeTopLidJoint.rotateX rightEyeBottomLidJoint.rotateX;

  setAttr "rightEyeJoint.rotateX" 30;
  setAttr "rightEyeTopLidJoint.rotateX" 15;
  setAttr "rightEyeBottomLidJoint.rotateX" 25;
  setDrivenKeyframe -cd rightEyeJoint.rotateX ...
     rightEyeTopLidJoint.rotateX rightEyeBottomLidJoint.rotateX;

  setAttr "rightEyeJoint.rotateX" 0;
}
```

11. That's it! If you now execute and run this procedure, Zuckafa's eye controller will realistically control the direction Zuckafa is looking in while deforming the surrounding skin and eyelids in a very realistic way. Because we did this through MEL, you have the power to change any part of the setup quickly and easily. Let's say, for example, that we encounter a problem where the eyeballs rotate too far. To decrease the range, it's as simple as editing a couple of values in the script and hitting the shelf button! This kind of flexibility is very handy, especially in a tight production.

12. Now that you understand how to script expressions and driven keys, it's a simple matter of going through each controller curve and writing the necessary MEL to connect them. For the majority of the controllers, the connections are very easy and straightforward. Open connectZuckafaFace.mel in a text editor to see the entire script used to connect *all* of the controllers.

13. Open exercise4.3_Start.mb. Open the connectZuckafaFace.mel script file and execute it. To connect all of his facial controls, simply execute 'connectZuckafaFace' from the command line. If no errors are encountered, the feedback line should spit out the following message:

```
Zuckafa's Face UI Connected.
```

14. Experiment with Zuckafa's facial UI to see how I chose to use each controller to affect the blendshapes and joints. Open exercise4.3_Finished.mb to see the finished, connected rig. Try posing Zuckafa to get different expressions like happy, sad, confused etc...

Connecting your shape set to a user interface can take time and patience. For Zuckafa, I chose to use a combination of set driven keys and expressions. You may choose to use only one or the other depending on your production needs. For cut-scene animation or film/television, it will not really matter what method is used. If your creature is expected to dynamically change expressions based on in-game AI, then you may need to stick to a connection method that the game engine supports. Regardless, as a TD, you should be always on the lookout for ways to make the setup process as painless as possible. Using a MEL script like I showed you here will help a lot.

MEL Scripting – The Creation of Zuckafa's Pose Library UI:

This section of chapter four will take a detour to concentrate on some of the issues involved in creating a MEL-based user interface for a facial rig. This is not to be confused with the curve based UI introduced earlier. This interface is an extra layer of control that will really help make the rig easy, fast and efficient.

For all but the most simple productions, animators will constantly be posing the creature's face over and over again. Giving the animators a

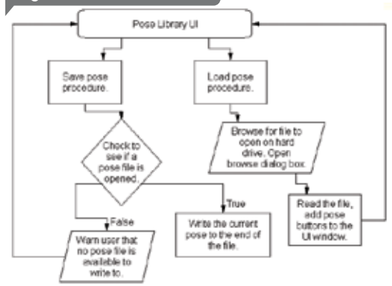
library of poses to choose from can really speed things up. Posing the face becomes as simple as clicking a button. Having the ability to save a specific pose will enable animators to create a collection of their most used poses. This collection or 'pose library' can then be saved to disk and shared amongst the studio.

Systems like these are employed in many major productions. Unfortunately, Maya does not natively provide a way of creating pose libraries. Of course, MEL will save the day by allowing us to program our own solution. Using a pose library can be very useful, but learning how to *program* one will open up a world of new possibilities. Programming a pose library involves, what I believe, to be the most overlooked feature in MEL, file input/output.

■ Zuckafa's Pose Library UI

As with every application building endeavor, it will help us to plan out exactly how the program should work before you even start coding. Throughout all of the MEL coverage in this book, I have been preaching about how important it is to plan everything out. In addition to pseudo code, you may find that a flowchart can help you in the planning phase. Check out FIGURE 4.146 to see the flowchart that describes the pose library functionality of the facial UI MEL script.

Figure 4.146.


Pose Library UI
Save pose procedure.
Load pose procedure.
Check to see if a pose file is opened.
Browse for file to open on hard drive. Open browse dialog box.
Read the file, add pose buttons to the UI window.
False
True
Warn user that no pose file is available to write to.
Write the current pose to the end of the file.

At the top of the chart, we have our user interface. This is a MEL window that contains, among other things, two buttons for saving/loading poses. The pose library works like this:

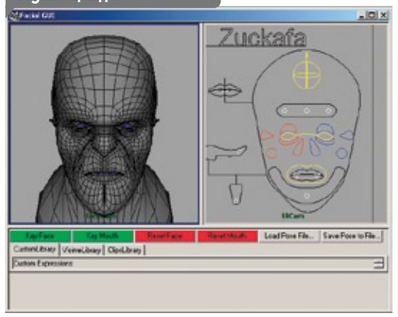
1. The user opens up Zuckafa's scene file and sources the facial UI. This brings up the window that contains all of the controls for Zuckafa's pose library system.

2. The user then loads Zuckafa's pose library by clicking on the 'Load Pose File' button. This brings up a dialog box that asks the user to find the pose library file. If no pose library currently exists, the user can start a new one by simply opening an empty text file.

3. With a pose library (or empty text file) loaded, the user can then start saving poses to the open file. To do this, the animator poses Zuckafa's face (using the curve controllers) and then hits the 'Save Pose to File' button. A dialog box will prompt the user to enter a name for the pose and specify an icon to represent it. A new button is created and added to the custom library tab.

4. The user can now click this new button to put Zuckafa's face into the recorded pose FIGURE 4.147 . The user can continue adding poses in this fashion to build up a library of available poses. When the animator closes down the program, the poses are still saved in the text file on the hard drive. This text file can be re-opened at a later date, or shared amongst the studio.

Figure 4.147.



If the user attempts to save a pose *before* he has loaded a library, the program should return a warning. Error checking like this is essential to ensuring that the system is stable enough for a production environment.

■ File Input/Output (I/O)

Up until now, all of the MEL scripts covered in this book have worked on some user-defined data or selection. They all follow the same format. That is, the user selects an object(s) and then executes a script. The script performs some action or series of actions on that object. This completes a specific task. It may be that this is all the functionality you may ever need. Indeed, other books that are solely dedicated to MEL scripting do not even include coverage of file input/output because they consider it to be something only used in the most *specialized* of circumstances. Other books consider file I/O something that is perhaps too difficult or extraneous to cover.

I couldn't disagree more. File I/O can be used in rigging systems, rendering systems, animation pipeline scripts and much more. To top it off, it is not difficult to implement. There is no doubt, reading and writing files can be tricky, but the benefits far outweigh the time it takes to learn.

This section will provide a nice introduction to reading and writing files. By the time we are done here, you will see exactly how to create your own custom file format for storing any information you want. This section will also show you how to write a parser to read a custom file format. In this way, we will build a system for saving and retrieving libraries of facial poses. By the end of this section, you will be in love with Maya's file I/O capabilities.

■ File Formats

You are already familiar with several different file formats, even if you do not know what they are. Windows applications use tons of different file formats to store all sorts of data, whether it be pictures, text, music or video. Maya itself supports two native file formats, .mb and .ma which stand for 'Maya Binary' and 'Maya ASCII'. A file format is simply a way of storing a specific type of data. Of course, Maya's formats are used for storing the information from a Maya scene file.

Whenever a programmer builds an application that saves out data, he/she must be aware of exactly how that data is to be formated. The format of a file describes the way data is organized in the file. File formats are very specific and must be carefully prepared. Knowing the order with which data is arranged in a file enables the programmer to write code that will read the data and use it appropriately. The order that the data appears in a file *is* the file's format.

The pose library system that we are building will be capable of writing out an ASCII text file that contains the exact translation and rotation of each of Zuckafa's facial controller curves. In this way, the file system records the facial pose to the hard drive. In order to read this pose library file, we must specify exactly how we want the data to be arranged in the file.

This brings up the problem of exactly what constitutes a pose? For the program are going to develop, a pose consists of three main pieces of data.

1. *The name of the pose:* The program works by creating a button, inside the window, for each pose. This allows the animator to simply hit a button to pose the face. The name of the pose is simply the name used for the button and will appear below the button's icon in the window. For example, a pose might be named happy, sad, angry etc...

2. *The location of the icon to represent the pose:* This is simply the directory path that tells the program where the image for the button is located on the hard drive. A typical directory path might look like this, 'C:/Documents and Settings/UsersName/My Documents/maya/6.0/prefs/icons/ZUCKAFA HAPPY.xpm'. This is the path to the image that will be used as an icon for the 'happy' pose. Happy.xpm is an image file with a picture of Zuckafa looking happy. An .xpm file is a cross platform compatible picture format that Maya uses for all of it's icons.

3. *The translate and/or rotate values of each controller curve:* The last piece of data that must be stored is the actual pose itself. This is comprised of a series of values that represent the translate and rotate values of each of Zuckafa's controller curves.

All three of these pieces of data combine to create a single *pose*. A series of poses will combine to create a *pose library*. When we write this data out, it will be stored in an ASCII text file. To verify that a pose file contains what it is supposed to, you can open it up with any ASCII text editor (like Mel Studio, Notepad or Emacs). If you want to see an example of a very advanced file format, try saving and opening a Maya ASCII (.ma) file in a text editor. The first lines of an .ma file look like this:

```
//Maya ASCII 6.0 scene
//Name: test.ma
//Last modified: Tue, Feb 01, 2005 04:43:23 PM
requires maya "6.0";
currentUnit -l centimeter -a degree -t film;
fileInfo "application" "maya";
fileInfo "product" "Maya Unlimited 6.0";
.......
```

Notice that each piece of the file tells something about the file. When you open a Maya ASCII file in Maya, a program will read through this file and effectively rebuild your scene from scratch. This is how you are able to return to a scene that looks exactly like it was before you last closed it.

We can learn how file formats, in general, work by observing the contents of an .ma file. If you have ever tried to open a Maya 6.0 file using version 5.5 or less, you will notice that Maya returns an error and will not let you continue opening the scene file. This is not uncommon as most applications have some sort of protection to prevent people from opening a file that may not work on their current version. Newer feature additions and enhancements cause this incompatibility issue.

You can see how the Alias engineers have prevented this backwards incompatibility issue by observing the contents of an .ma file. Line four of the .ma file specifies what version of Maya is required to open the file. If the current version is less than the required version number found in the file's fourth line, an error is returned.

Notice that on line five, the .ma file specifies the scene units (centimeters). The program that reads .ma files will look for exactly this information on exactly line five. Because the .ma file format follows strict rules, the program will always find the information it expects on the exact line it expects. The .ma file continues on in this manner describing the saved scene file for the program to rebuild.

Just like an .ma file, our file will hold text information to describe data. Instead of a scene file, ours is a pose file. Our file format is much more simple but it uses the same basic concept. When arranged into it's text file, a pose file can look like this:

```
happy
C:/Documents and Settings/User/.../icons/Happy.xpm
setAttr mouthIntensity.tx 1;setAttr jawOrient.tx 0; .....
sad
C:/Documents and Settings/User/.../icons/Sad.xpm
setAttr mouthIntensity.tx .5;setAttr jawOrient.tx .25; .....
```

This particular pose file contains two poses, happy and sad. Notice that each pose includes the three components mentioned earlier (name of pose, location of icon, position of controller curves). Each of these components is on it's own line in the file. This is important as our system will read every three lines in the file as a single pose. As the program reads through the file, it will create a button for each pose. The first line represents the name of the button. The second line is used to add a picture icon to the button. The third line is used as the actual *command* that the button will execute when pressed. This is why the MEL command, 'setAttr' (setAttr means 'set attribute'), is used. We want the button to set the attributes of the controller curves on Zuckafa's facial rig to match the recorded pose.

This is a simple format, but it must be fully understood in order to make sense of the next part of this section where I explain the actual MEL that will make all of this happen. I hope that as you begin to understand file formats, you try and augment this one to create a format that is more robust. Perhaps you could add a character check to ensure that a pose library is not loaded for the wrong character. Or maybe you could add a line to the format that specifies which animator created the pose. The possibilities are endless.

■ Getting Started With the Interface

This project, like all of the scripts in this book so far, is all included in one single MEL script file. I could have split up the procedures into separate files, but I did not feel this would be necessary. In the interest of simplicity, I've kept them all together. Whenever you start a project like this, recall that it is absolutely essential to have the code for the interface at the *bottom* of the script file. This is because the interface contains the buttons which call all of the other procedures. If the proce-

dures are declared after the interface code (in the script file), the code will generate an error when sourced.

A procedure must be declared ***before*** it is called.

So to start off with, let's quickly cover the new UI code. The user interface for this script incorporates a new layout called the pane layout (paneLayout). A pane layout is a handy way of splitting up a window into separate panes. Through the use of the '-panesize' and '-configurations' flags, the user can specify how the window should be split up and what size the panes should be. Panes are a great way of sectioning a window into individual pieces.
The pane layout is the first child of the window. To start off this script, make a generic window and placed the pane layout directly underneath it.

```
global proc zuckafaFacialUI ()
{
        if (`window -q -ex creatureUI`) deleteUI creatureUI;
        window -w 650 -h 600 -title "Facial GUI" creatureUI;
        paneLayout -paneSize 3 100 30 -configuration "top3" ...
                                        "facialPaneLayout";
        showWindow creatureUI;
}
```

This window code must be encompassed by a *global* procedure. This will enable us to embed the UI directly into a Maya scene file. This will be covered later.

The first line in the procedure should be familiar. This 'if' statement prevents multiple iterations of the same window from being created at the same time. If the window named 'creatureUI' already exists, the if statement evaluates to true and the window is deleted. Then the rest of the procedure procedes to rebuild it.

The second line in the procedure uses the window command to create a window with a width of 650 pixels and height of 600. This should make the UI large enough for most monitor resolutions. The window is given a title of 'Facial GUI' and a name of 'creatureUI'.

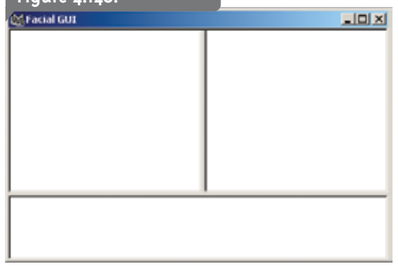
The third line is the interesting one. This creates the pane layout. The '-panesize' flag can be used to specify the size of a pane. In this case, we want the third pane to have a width of 100 percent and a height of 30 percent. Notice that I did not specify the size in *pixels* but rather in *percent*. The panes in a pane layout will automatically resize to accommodate the size of the parent (in this case the window is the parent). This is especially important for this interface because the animator may wish to resize the window to accommodate his/her workspace.

The '-configuration' flag specifies the type of configuration as described in the MEL command reference. The "top 3" configuration makes three separate panels, two of which rest on top of the third. There are fifteen other configurations that can be used with the pane layout. Refer to the MEL command reference when you need to choose one. The 'top3' configuration was needed for this particular instance because it will enable us to organize the facial user interface into three separate sections. These three sections are:

1. A viewport that looks through a camera that is facing Zuckafa's face.
2. A viewport that looks through a camera that is facing Zuckafa's curve controllers.
3. A place to house the pose library.

If we source and run this program now, we will get a window that is separated into three different sections FIGURE 4.148. If you try to scale the window, the sections will also scale to accommodate the window.

Figure 4.148.

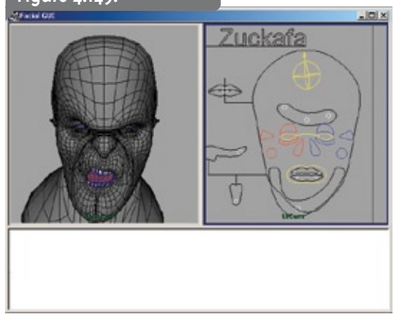


That is all fine and dandy but we want these sections to actually house some useful stuff. In the case of the top two sections, they need to actually look through a Maya camera and display an openGL viewport. MEL has a really easy way of doing this. The 'modelPanel' layout will display what is called a 'modelEditor'. This is a UI element that behaves exactly like a Maya viewport. We can even specify what camera we want it to 'look through'. Add the following lines of code below the pane layout:

```
// THIS IS THE FACIAL INTERFACE PANEL
modelPanel -cam "faceCam" -menuBarVisible 0 -l ...
                                    facialInterfacePanel;
setParent..;

// THIS IS THE FACIAL CAMERA PANEL
modelPanel -cam "UICam" -menuBarVisible 0 -l facialCamPanel;
//Set the heirarchy back one level
setParent..;
```

These two modelPanel commands create a Maya viewport inside each of the two top panels FIGURE 4.149. The camera that each of the viewports looks through is specified by the '-cam' flag. I had to make sure to include the 'faceCam' and 'UICam' in the Zuckafa rig scene file. These cameras are pointed at the face and the UI respectively. I also tacked on the '-menuBarVisible' flag to say that I did *not* want the menu bars to be visible. With the menu bar, the animator could potentially mess the viewports up so that they would not look through the correct cameras anymore. This can easily be avoided by simply hiding the menu. Animators are a pesky breed, they will try everything they can to screw up your hard work.

Figure 4.149.

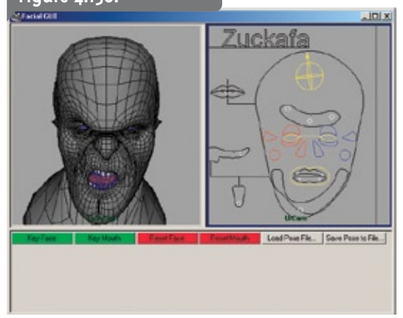


Notice the use of the setParent commands. This sets the hierarchy back a notch. We are now ready to specify exactly what we want in the last pane. This is the bottom pane.


```
columnLayout ;
 rowColumnLayout -nc 6;
 //Key Face Button
 button -bgc .1 .8 .1 -c "keyFace;" -label "Key Face";
 //Key Mouth Button
 button -bgc .1 .8 .1 -c "keyMouth;" -label "Key Mouth";
 //Reset Face Button
 button -bgc .94 .035 .164 -c "resetFace;"-label "Reset Face";
 //Reset Mouth Button
 button -bgc .94 .035 .164 -c "resetMouth;"-label "Reset Mouth";
 //Load Pose File Button
 button -label "Load Pose File..." -c ("loadPoseLibrary");
 //Save Pose to File Button
 button -label "Save Pose to File..." -c ("addPoseToLibrary");
setParent..;
```

The use of the column and rowColumn layouts should be nothing new. The rowColumn layout is given six columns as specified by the '-nc' flag (-nc is -numberOfColumns in long form). Each column has it's own button. Each of these buttons call their own procedures. These buttons provide the core functionality of the user interface FIGURE 4.150. Of course, these procedures need to be declared before we can call them. When we are done with our dissection of the user interface code, the rest of this chapter will be dedicated to covering these procedures. After all of the buttons, the setParent command sets the hierarchy back to the columnLayout.

Figure 4.150.



To finish the interface, we are going to create a scroll layout to house the buttons for the pose library. You may have noticed that the finished pose library is under it's own tab. This tab layout is used to separate the custom pose library from other libraries that you may wish to include. Zuckafa has a tab for full facial poses, visimes and clips. Visimes and clips will be explained later.

Regardless, the tab and scrollLayouts can be added directly under the column layout to place the libraries under the row of buttons we created with the last snippet of code.

```
tabLayout;
   // Expression Library Tab
   scrollLayout -horizontalScrollBarThickness 0 "CustomLibrary";
        //Row layout to house pose buttons.
        rowColumnLayout -nc 6 "expressionLibraryLayout";
          text "Custom Expressions: ";
          text " ";
          text " ";
          text " ";
          text " ";
          text " ";
         setParent..;
```

```
//Set hierarchy back to the tab layout.
setParent..;
```

Remember, we are building three separate tabs. This code will create the first one. This is the pose library tab. It is a scroll layout that will house all of the buttons used to pose the face. I set the '-horizontalScrollBarThickness' flag on the scroll layout to '0'. This effectively turns off the horizontal scroll bar. In this case, the horizontal scroll bar is useless and just takes up space. Always be on the lookout for ways to make your interfaces more space efficient.

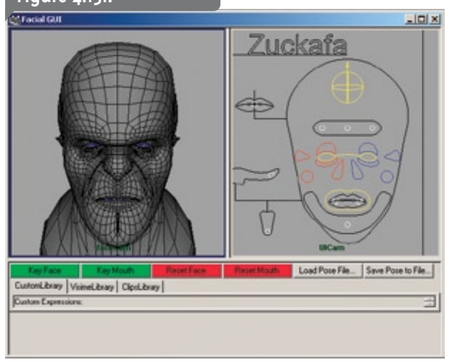
The scroll layout is then given a row/column layout as a child. This will arrange all of the pose buttons into rows. Because the script must dynamically add the pose buttons by reading a pose file, no buttons are added explicitly. The six empty text commands just fill up the first row in the row/column layout so that the first button added is in the second column. This is just a style issue.

Because we added a couple of setParent commands, the hierarchy is set back to the tab layout. We can safely add the other two tabs.

```
// Visime Library Tab
scrollLayout -horizontalScrollBarThickness 0 "VisimeLibrary";
  rowColumnLayout -nc 4 "visimeLibraryLayout";
    text "Visimes: ";
  setParent..;
setParent..;
// Clips Library Tab
scrollLayout -horizontalScrollBarThickness 0 "ClipsLibrary";
  rowColumnLayout -nc 6 "clipsLibraryLayout";
    text "Clips: ";
```

These two tabs are exactly the same as the expressions tab. They contain a scroll layout with a row/column layout underneath it FIGURE 4.151. Consider these tabs placeholders for future additions. I'll explain this later.

Figure 4.151.



This completes the interface procedure. There are a total of six other procedures. Each one of these is called by one of the buttons we added to the interface. Before I get into the pose library procedures and the file I/O algorithms they use, I want to get the other buttons out of the way.

■ The 'Set Key' and 'Reset' Buttons

The first four buttons are dedicated to reseting and keying the mouth or the entire face. These four buttons call four different procedures. The buttons and there procedures are as follows:

1. Key Face: This button calls the 'keyFace' procedure. This procedure sets a key for every controller on the face.
2. Key Mouth: This button calls the 'keyMouth' procedure. This procedure will set a key for every controller curve that affects the mouth. This is handy for fleshing out lip sync. It is not uncommon to animate the mouth separately from the eyes.
3. Reset Face: This button calls the 'resetFace' procedure. This procedure simply sets all of the attributes for all of the controllers to zero. It is like setting the bind pose for the face rig.
4. Reset Mouth: This button calls the 'resetMouth' procedure. This procedure resets the mouth controllers to their defaults.

The procedure declarations for these buttons are nothing new. To set keys, the procedures use the 'setKeyframe' MEL command. The keyMouth procedure looks like this:

```
//Will set a key on the mouth controllers only
proc keyMouth()
{
  setKeyframe -breakdown 0 -hierarchy none -controlPoints ...
     0 -shape 0  {"mouthSync", "lowerLip", "upperLip", ...
      "jawOrient", "tongue", "smileFrown"};
}
```

All of the flags on this setKeyframe command are actually just the defaults. You needn't include them if you do not want to. The list of objects to keyframe is enclosed by the curly braces. For the 'keyFace' procedure, this list includes all of the other controllers as well.

The other two buttons are for *reseting* the controllers. To reset the controller curves, we can use the setAttr MEL command. To create this procedure, I gathered a list of all the attributes on the facial rig, then I wrote them out with the setAttr command to set them all to zero. This returns them to their defaults.

```
//Will reset the mouth controllers to zero
proc resetMouth()
{
        setAttr mouthIntensity.tx 0;
        setAttr jawOrient.tx 0;
        setAttr jawOrient.ty 0;
        setAttr upperLip.tx 0;
        setAttr upperLip.ty 0;
        setAttr lowerLip.tx 0;
        setAttr lowerLip.ty 0;
        setAttr tongue.tx 0;
        setAttr tongue.ty 0;
        setAttr tongue.inOut 0;
        setAttr cheeks.tx 0;
        setAttr cheeks.ty 0;
        setAttr smileFrown.tx 0;
        setAttr smileFrown.ty 0;
        setAttr mouthSync.tx 0;
        setAttr mouthSync.ty 0;
}
```

The 'resetFace' procedure looks exactly the same except it includes all of the controllers, not just the mouth ones.

This concludes the reset face and key face procedures. They really are very simple, but can help the animator a lot. It is worth the ten minutes it takes to add them.

Coding the Pose Library

The last two buttons on Zuckafa's facial UI are used to open pose files and add poses to a file. These procedures will introduce you to the basics of file I/O.

To start using a pose library, the user must first open a pose file. If the user wants to start a new pose file, they can simply load an empty text file. It would seem logical to explain how the 'load file' procedure works before explaining the 'add pose' procedure. But the way the file is loaded is heavily dependent on the way it is formatted when written. For this reason, I'll explain the 'addPoseToLibrary' procedure first.

When the user clicks the 'Add Pose to Library' button, the 'addPoseToLibrary' procedure is called. This procedure first checks to see if a library file has been loaded. If it has not, a warning is returned and nothing else is done. To keep track of whether or not a pose library has been loaded, I chose to use a global string variable, $poseLibraryInUse. This variable houses the directory path for the currently used library file. If no library file has been loaded yet, the string will simply contain the character 'X' (it is initialized to 'X'). So when we are checking to see if a library file has been loaded before adding a new pose, the program can check this global variable to see if it contains something other than 'X'. In MEL, it looks like this:

```
// Stores the directory location of current library file in use.
global string $poseLibraryInUse;
//Intialize to 'X'.
$poseLibraryInUse = "X";

//With the global $poseLibraryInUse variable declared, we can
//start the 'addPoseToLibrary' procedure.

//Add pose to the currently loaded library procedure.
proc addPoseToLibrary ()
{
  // Current library that is loaded.
  global string $poseLibraryInUse;

  //Check to see if the user has loaded a library.
  if ($poseLibraryInUse == "X")
  {
    //Warn user if no library is loaded.
    warning "You must load a library before saving new poses.";
  } else
  {
    //Write the pose to the file.
  }
}
```

Notice that it appears that I have declared the $poseLibraryInUse string both inside and outside the procedure. I also included the word 'global' in front of this string declaration. This is necessary to allow this procedure to access this variable. While global variable can be useful, be warned, heavy use of global variables can lead to nasty bugs. Because global variables can be accessed and edited anywhere in your program, they can be difficult to predict. I recommend only using them wherever necessary. I had to make this string a global so that it could be accessed by other procedures. A non-global or 'local' variable only exists within the scope of the procedure it is declared in. For this case, a local variable would not have worked.

The first thing this procedure does is check to see if $poseLibraryInUse contains a path to an opened file. If the $poseLibraryInUse variable does indeed contain a path to an opened file, then we can begin writing our pose to that file. If $poseLibraryInUse only contains an 'X', then we warn the user and exit out of the procedure. If it does contain a path, we can continue with the writing.

To start with, we need the user to specify a name for the pose. To do this, why not bring up a nice dialog box? MEL has a very easy to use little dialog that can prompt the user to input a string. The following code should be added in the 'else' section of the conditional statement from the last snippet:

```
//Prompt user for name of pose
promptDialog -message "Enter name of pose:" -button "Ok" ...
                                            -defaultButton "Ok";
string $nameOfPose = `promptDialog -query -text`;
```

The 'promptDialog' command brings up a small dialog box that can display a small message with an 'Ok' or 'Cancel' button FIGURE 4.152. The '-message' flag specifies what message is displayed in the prompt box. The '-button' flag specifies the name that will be printed on the button and the '-defaultButton' flag determines which buttons will be selected by default (the button that will be pressed if the user hits the Enter key).

Figure 4.152.

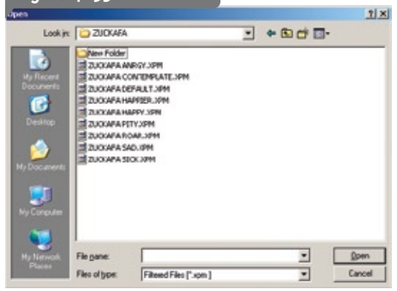
After calling the dialog, the user enters a name for the pose and presses 'Enter'. This stores the input into the $nameOfPose variable. To do this, I queried the '-text' flag which returns the user input from the last prompt dialog. Notice the entire query command is enclosed in back ticks (``). This ensures that the string variable is equal to the *returned* value of the command and not the literal command itself.

Recall that we earlier defined a pose as being a name, an icon and a series of 'setAttr' commands. We now have the name, so let's continue with finding an icon for the pose. To do this, we could have the user specify an exact directory location inside another prompt dialog. This, however, would be a very difficult way of specifying an icon because the user would need to know the exact path and filename of the icon. To avoid this, we can use a browse dialog.

You have no doubt been exposed to thousands of browse dialogs. These little windows act as a conduit to the directory system on your computer's hard drive or a network hard drive. When you use `File > Open Scene...` in Maya, you are using a file browsing dialog box. We can use one of these to help our user find the location of the icon they want to use for their new pose. After entering the name of the pose, another dialog box will pop up, this will allow them to find the icon they want to use FIGURE 4.153.

This code can be placed directly after the last snippet:

Figure 4.153.




```
//Prompt user to find icon for pose

//Get the current project folder directory
$currentProject = `workspace -q -rd`;
$currentProject = ($currentProject + "*.xpm");

//Prompt user to find the file to load
string $iconLocation = `fileDialog -dm $currentProject`;
```

The first two lines in this snippet are used to find the currently used project directory. The project directory is used as the default starting place where the file browsing dialog box will point to. Recall that Maya's project folders are used to house all of the different pieces of an animation project (textures, .mb files, cache data etc...). It would only seem natural to have the icon images in this project folder as well. So to help the user, we can determine where the current project folder is and start the file browser at that directory. The 'workspace' MEL command can be used to access all sorts of information about the user's current workspace. With the '-rd' (-rootDirectory in long form) flag attached, the workspace command will return the root directory of the current project folder.

The second line concatenates the string "*.xpm" onto the end of the $currentProject variable. The '*.xpm' is a wild-carded file specifier. This will cause the file dialog box to display only files of the .xpm type. In case you were wondering, .xpm is the picture file format that Maya uses for all of it's icons.

Unfortunately, creating xpm images cannot be done with most photo editing software (Adobe Photoshop included). To create .xpm image files, I use a freely available tool from highend3d.com by the name of 'makexpm'. This tool will convert any bitmap file to the .xpm format. You will need this tool if you want to make custom icons for Maya.

With the project directory found and the *.xpm filter tacked on, we are ready to call the dialog box. Unlike the 'promptDialog' command that brings up a simple text input, the 'fileDialog' command brings up a full fledged browser. The '-dm' (-directoryMask in long form) flag is used to specify where we want the file dialog to point to when it is first opened. In this case, the project directory root. Notice that the result of the fileDialog command is returned into the $iconLocation string variable. Again, this is done with the infamous back ticks (``). After this line, the $iconLocation string will house the exact location of an .xpm file that the user chose.

We now have two of the three major pieces of data that we need to complete the pose. You may recall that the last thing we need is the actual pose information itself. This is comprised of the attributes and their

values for every controller on Zuckafa's facial rig. I made a complete list and determined that there are 36 different attributes that must be recorded. This includes his eyes, tongue, mouth, head and everything else.

Recall that the only reason we are storing this information is so that we can later retrieve it to build a button that the animator can press to pose the face. Affecting an attributes value with MEL is done through the setAttr command. Knowing this, we can store the current facial pose as a series of setAttr commands. This way, when we go to load the pose from the file, we can simply load this concatenated series of setAttr commands directly into the command of the button itself. So when the button is pressed, it sets the attributes of all the controllers according to what we saved.

We are going to store all of the setAttr commands into one single, very long, string variable. To do this, let's declare a string to house all of the commands. We also needs an array of floats to hold the values of all of the different attributes on the facial rig.

```
//Construct button command
string $poseButtonCmd;
float $sliderValues[];
```

Each index in the $sliderValues array will house the value of an attribute. To get the individual values, we can use the 'getAttr' command. This MEL command will return the value of any attribute you specify. Assign each index in the array to one of the attributes on the facial rig.

```
//Put all slider values in a float array
$sliderValues[0]= `getAttr mouthIntensity.tx`;
$sliderValues[1]= `getAttr jawOrient.tx`;
$sliderValues[2]= `getAttr jawOrient.ty`;
$sliderValues[3]= `getAttr upperLip.tx`;
$sliderValues[4]= `getAttr upperLip.ty`;
$sliderValues[5]= `getAttr lowerLip.tx`;
$sliderValues[6]= `getAttr lowerLip.ty`;
$sliderValues[7]= `getAttr tongue.tx`;
$sliderValues[8]= `getAttr tongue.ty`;
$sliderValues[9]= `getAttr tongue.inOut`;
$sliderValues[10]= `getAttr cheeks.tx`;
$sliderValues[11]= `getAttr cheeks.ty`;
$sliderValues[12]= `getAttr smileFrown.tx`;
$sliderValues[13]= `getAttr smileFrown.ty`;
$sliderValues[14]= `getAttr snarl.tx`;
$sliderValues[15]= `getAttr snarl.ty`;
$sliderValues[16]= `getAttr mouthSync.tx`;
$sliderValues[17]= `getAttr mouthSync.ty`;
$sliderValues[18]= `getAttr eyeLookAt.tx`;
$sliderValues[19]= `getAttr eyeLookAt.ty`;
$sliderValues[20]= `getAttr rightEyeTopLid.ty`;
$sliderValues[21]= `getAttr leftEyeTopLid.ty`;
$sliderValues[22]= `getAttr rightEyeBottomLid.ty`;
$sliderValues[23]= `getAttr leftEyeBottomLid.ty`;
$sliderValues[24]= `getAttr leftEyeBottomLid.squint`;
$sliderValues[25]= `getAttr leftBrow.ty`;
$sliderValues[26]= `getAttr rightBrow.ty`;
$sliderValues[27]= `getAttr middleBrow.ty`;
$sliderValues[28]= `getAttr middleBrow.squeezeBrow`;
$sliderValues[29]= `getAttr leftEyeLookAt.tx`;
$sliderValues[30]= `getAttr leftEyeLookAt.ty`;
$sliderValues[31]= `getAttr rightEyeLookAt.tx`;
$sliderValues[32]= `getAttr rightEyeLookAt.ty`;
$sliderValues[33]= `getAttr head.rx`;
$sliderValues[34]= `getAttr head.ry`;
$sliderValues[35]= `getAttr head.rz`;
```

Whew! That's a long list. With that, the $sliderValues array contains 36 different values. This is every attribute on the facial rig. We can now use this array to build the $poseButtonCmd. To do this, we will concatenate the 'setAttr' command, followed by the name of the object, followed by the value of the attribute (from the $sliderValues array). So the basic format is as follows:

```
Pose Button Command = "setAttr" + "name of object.attribute" +
"value of attribute"... and so on for each controller...
```

These three strings are added together for each attribute in the facial rig. When they are all added up, the single command stored in $poseButtonCommand will set the value for every attribute on the whole facial rig. To compute these concatenations, we need to make a full list that adds all of the different components together.

```
//Construct the string
$poseButtonCmd = ("setAttr jawOrient.tx "+$sliderValues[0]+";")
+ ("setAttr mouthIntensity.tx " + $sliderValues[1] + ";")
+ ("setAttr jawOrient.ty " + $sliderValues[2] + ";")
+ ("setAttr upperLip.tx " + $sliderValues[3] + ";")
+ ("setAttr upperLip.ty " + $sliderValues[4] + ";")
+ ("setAttr lowerLip.tx " + $sliderValues[5] + ";")
+ ("setAttr lowerLip.ty " + $sliderValues[6] + ";")
+ ("setAttr tongue.tx " + $sliderValues[7] + ";")
+ ("setAttr tongue.ty " + $sliderValues[8] + ";")
+ ("setAttr tongue.inOut " + $sliderValues[9] + ";")
+ ("setAttr cheeks.tx " + $sliderValues[10] + ";")
+ ("setAttr cheeks.ty " + $sliderValues[11] + ";")
+ ("setAttr smileFrown.tx " + $sliderValues[12] + ";")
+ ("setAttr smileFrown.ty " + $sliderValues[13] + ";")
+ ("setAttr snarl.tx " + $sliderValues[14] + ";")
+ ("setAttr snarl.ty " + $sliderValues[15] + ";")
+ ("setAttr mouthSync.tx " + $sliderValues[16] + ";")
+ ("setAttr mouthSync.ty " + $sliderValues[17] + ";")
+ ("setAttr eyeLookAt.tx " + $sliderValues[18] + ";")
+ ("setAttr eyeLookAt.ty " + $sliderValues[19] + ";")
+ ("setAttr rightEyeTopLid.ty " + $sliderValues[20] + ";")
+ ("setAttr leftEyeTopLid.ty " + $sliderValues[21] + ";")
+ ("setAttr rightEyeBottomLid.ty " + $sliderValues[22] + ";")
+ ("setAttr leftEyeBottomLid.ty " + $sliderValues[23] + ";")
+ ("setAttr leftEyeBottomLid.squint " + $sliderValues[24] + ";")
+ ("setAttr leftBrow.ty " + $sliderValues[25] + ";")
+ ("setAttr rightBrow.ty " + $sliderValues[26] + ";")
+ ("setAttr middleBrow.ty " + $sliderValues[27] + ";")
+ ("setAttr middleBrow.squeezeBrow " + $sliderValues[28] + ";")
+ ("setAttr leftEyeLookAt.tx " + $sliderValues[29] + ";")
+ ("setAttr leftEyeLookAt.ty " + $sliderValues[30] + ";")
+ ("setAttr rightEyeLookAt.tx " + $sliderValues[31] + ";")
+ ("setAttr rightEyeLookAt.ty " + $sliderValues[32] + ";")
+ ("setAttr head.rx " + $sliderValues[33] + ";")
+ ("setAttr head.ry " + $sliderValues[34] + ";")
+ ("setAttr head.rz " + $sliderValues[35] + ";");
```


After this, the $poseButtonCmd will house the code that will be executed when the pose button is pressed by the animator. When I was constructing this list of concatenations, I was careful to ensure that each 'object. attribute' was associated with the correct index from the $sliderValues array. Remember, each value in the array was initialized to a specific value on a controller curve's attribute. If we were to now print out the $poseButtonCmd string, it might look something like this:

```
setAttr mouthIntensity.tx 1;setAttr jawOrient.tx 0;setAttr jawOrient.ty -0.156; .... and this would continue on with a total of 36 different setAttr commands.
```

■ Writing the Pose to Disk

We now have all three of the pieces of data needed to write this pose to disk. This is the fun part. File I/O, in MEL, is very straightforward, but there are a couple of things that you must be aware of in order to avoid some common pitfalls.

As with all programming languages, before you can write data to a file, you must *open* the file. In MEL, this is done with the 'fopen' command. The 'fopen' command will take a string that represents the directory location and file name to open. In our case, we will feed it the $poseLibraryInUse string variable. This global variable is initialized in the load pose library procedure.

There are three different ways to open a file with MEL:

1. Writing (w): This method will cause any newly written data to overwrite any data that might already be in the file. The 'w' method starts writing at the beginning of the file.
2. Append (a): This method will place any newly written data at the *end* of the file. This can be used to add to an existing file.
3. Read (r): Opening a file in 'read' mode is done to get input from a file. This is used in the load procedure.

In order to open our pose library file to write to, we are going to use the 'append' mode. This will allow us to *add* the pose to the current file without overwriting existing poses that may already be in the file.

```
//Write new pose to file
float $fildID;
$fileID = fopen($poseLibraryInUse,"a");
fprint($fileID,($nameOfPose + "\r\n"));
fprint($fileID,($iconLocation + "\r\n"));
fprint($fileID,($poseButtonCmd + "\r\n"));
fclose($fileID);
```

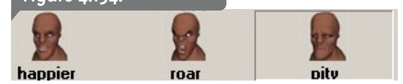
These six lines of code are all that is needed to write the pose to a file. Notice that I declared a new float variable called $fileID. When opening a file, the operating system returns a file id number. This number is used to uniquely identify a file on the disk. The fopen command returns the file id of the file we are opening into the $fileID float variable. Now we can use the $fileID variable wherever we want to edit the currently opened file. Notice that I tacked on the string "a" to the fopen command. This puts the file in 'append' mode as we discussed earlier.

Now, to actually print something to the opened file, we use the 'fprint' command. This command needs a fileID and some data. These two arguments are supplied by the $fileID variable and one of the three strings we constructed earlier. Notice that each 'fprint' command is followed by "+rn". Adding the 'r' and 'n' cause the printer to move to the cursor to the next line after writing. This is the same as hitting the enter key in a word processor. In this way, we can organize our file so that each piece of information is located on it's own line. Writing each piece of data to it's own line will be essential when we try to read the data in.

The last line uses the 'fclose' command. This command takes a float argument to specify the file id to operate on. The 'fclose' command closes the file. It is good programming practice to close a file after opening it. If you do not close the file, the user will be unable to edit the file again until it is closed or you shutdown Maya. Closing a file relinquishes control of the file, giving it back to the operating system.

The add pose procedure is almost finished now. The last task we need to program is that of actually adding a button to the interface. Adding a button in this procedure will prevent the user from having to re-open the file to see the a newly added pose button in the user interface. To represent a pose, I chose to use the 'iconTextButton'. This MEL command creates a button with an icon and some text that displays the name of the button FIGURE 4.154 . This is better for representing a pose than the default button that does not include support for an icon.

Figure 4.154.




```
//Create add-pose button
iconTextButton -parent "expressionLibraryLayout" -style ...
               "iconAndTextVertical" -image1 $iconLocation ...
                  -label $nameOfPose -command $poseButtonCmd;
```

Adding a button to an already existing interface is exactly the same with the exception that you must use the '-parent' flag. This flag will allow you to specify the name of the layout that you want the button to appear in. If you recall from the UI section, I named the pose library layout ' expressionLibraryLayout'. While you do nOt have to explicitly define a name for your layouts, doing so enables you to dynamically add elements to them as you need to. You can also query them.

The '-imagei' flag specifies what to use as an icon, the '-label' and the '-command' flags are assigned to their corresponding string variables that we constructed earlier. This will create a button with the proper icon, name and command to pose Zuckafa's face.

The last thing we need to do is print out some feedback for the user and close off the entire procedure with an end curly brace.

```
print ("Pose Added To : " + $poseLibraryInUse + "\n");
}//End of add pose procedure.
```

■ Load Pose Library File Procedure

Now that we have seen how to write to a file, this procedure will show you how to build a parser to read from your custom file format. This procedure, when called, must open a file and read each line in the file. As it reads through the file, we need the program to build a button for each pose and place this button into the proper layout (expressionLibraryLayout).

To load a library file, this procedure must follow some simple steps:

1. Firstly, we must find the current project directory. This information can then be used to provide a starting place for the user to browse for a pose file to open.
2. Next, we need to actually bring up the file dialog. This window will provide a nice way for the user to search for a file to open.
3. After selecting the file, we need to open it. This is done with the 'fopen' command.
4. Now we enter into a loop. This loop iterates through each line of the file, storing the contents of the line into a string variable. At the end of each iteration, these strings are used to construct a button that is added into the expression library layout.
5. After reading the file and creating the buttons, we can close the file and print some nice feedback for the user.

To get started, we declare the procedure and prompt the user for a file to open:

```
proc loadPoseLibrary ()
{
        //This string will be used later in the procedure.
        global string $poseLibraryInUse;

        //Get the current project folder directory
        $currentProject = `workspace -q -rd`;
        $currentProject = ($currentProject + "*.txt");

        //Prompt user to find the file to load
        string $fileLocation = `fileDialog -dm $currentProject`;

        //Open the file for reading ("r")
        print ("Opened : " + $fileLocation + "\n");
        $fileID = `fopen $fileLocation "r"`;
}
```

Much of this code should look familiar. The $currentProject string is initialized here in the same way as the add pose procedure. By querying the root directory (-rd) on the workspace command, we get the exact directory path we need. Whereas before we used this method to allow the user to search for an icon (.xpm), this time we are using it to search for an ASCII text file (.txt).

The $fileLocation string is initialized to the path of the file that the user chooses. The 'fopen' command is then used to open the file and prepare it to be read. Remember, just like this book, a file must be opened before you can read it.

The $fileID float variable will store the file id for the opened file. This is what the 'fopen' command returns.

Now we get into the meat and potatoes of the parser. To parse through the file, I will use one while loop. This loop will continue to iterate through the lines in the file until we get to the end of the file.

```
//Step through each line in the file and create necessary
//buttons.
string $nextLine = `fgetline $fileID`;

while (size($nextLine) > 0)
{
  int $i = 1;
  string $currentPoseName;
  string $currentIconName;
  string $currentButtonCmd;

  for ($i=1; $i<=3; $i++)
  {
    if ($i==1)
    {
      $currentPoseName = $nextLine;
      $currentPoseName = `substring $currentPoseName 1 ...
                                  (size($currentPoseName) - 1)`;
    } else if ($i==2)
    {
    $currentIconName = $nextLine;
    $currentIconName = `substring $currentIconName 1
                                  (size($currentIconName) - 1)`;
    } else if ($i==3)
    {
      $currentButtonCmd = $nextLine;
      $currentButtonCmd = `substring $currentButtonCmd 1
                                 (size($currentButtonCmd) - 1)`;
    }
      $nextLine = `fgetline $fileID`;
  }
iconTextButton -parent "expressionLibraryLayout" -style ...
      "iconAndTextVertical" -image1 $currentIconName -label ...
                          $currentPoseName -c $currentButtonCmd;
}
```

■ How the Parser Works

This is the most complicated part of the program. To understand how the loop is iterating through the file, you really only need to understand one new command. This is the 'fgetline' command. 'fgetline' is a special command that reads in *one entire line*. After reading the entire line of the file, the 'fgetline' command will knock the cursor down a line (like pressing the down arrow key on the keyboard).

Remember, our file is organized into groups of three lines. Each pose is represented by data stored on three separate lines. Because of this, our parser has a for loop nested inside the while loop. This for loop will iterate through three lines at a time before exiting. We store each line into one of three string variables, $currentPoseName, $currentIconName and $currentButtonCmd. You may notice that while the returned string from $getline is stored into each of these variables, there is another line of code that appears to edit that information afterwards:

```
$currentIconName = $nextLine;
$currentIconName = `substring $currentIconName 1 ...
(size($currentIconName) - 1)`;
```

These lines are necessary to strip the last character from the string. When reading in from a file, you may find that the input has an extra null character attached to it. This null will cause problems if it is not removed. The substring command will remove characters outside of the specified range. In this case, we just want to remove the last character (the null) so we specify a range from 1 to n-1 where n is the size of the string. To find the size of the string, we can simply use the 'size' command.

After iterating through each of the three lines, we break out of the for loop, create our button and go back through the while loop. This is done as many time as is necessary until we hit the end of the file. If the fgetline command is called when the cursor is at the end of the file, it will return 0. When this happens, we break out of the main while loop and return back to the rest of the procedure.

If you are still having trouble understanding how these loops work to read in our file format, remember a couple of important points:

1. Each pose is stored on three separate lines. The for loop iterates through three lines at a time and stores the *contents* of each line into a separate variable.
2. The 'fgetline' command returns the *contents* of an *entire* line. After reading the line, it is important to remember that the fgetline command places the cursor at the beginning of the next line in the file.
3. The iconTextButton is created in exactly the same way as described for the add pose procedure.

■ Wrapping it all up

After the while loop exits, at the end of the file, we have a couple more things to do. Recall that we must close every file that we open. This is to ensure that the operating system can take control of the file again. We also need to set our $poseLibraryInUse variable to equal the directory path to the file we just opened. This is so that the program can keep track of which file to add poses to.

```
//Close the file.
fclose $fileID;
//Enlarge the scrollLayout to make room for the buttons.
scrollLayout -e -h 100 ("CustomLibrary");
//Print some feedback for the user.
print "Pose Library Loaded!\n";
//Set the global variable to keep track of the currently used
//file.
$poseLibraryInUse = $fileLocation;
} //End of load pose file procedure.
```

This concludes the load pose procedure. After the procedure is finished, Zuckafa's user interface will contain all of the new buttons that were stored into the pose library file. There is an included pose library on the DVD that contains nine different poses (ZuckafaFacialPoses.txt). Try loading this file and observe the new buttons as they get added to the interface. I also made some nice icons for these buttons to use. These icons are also included on the DVD.

■ Finishing Touches

Having been exposed to MEL's file I/O capabilities, I hope that you have already begun to see how powerful they are. The ability to read and write your own files can really save the day. This kind of programming knowledge can be used for hundreds of different things. Video game companies often use MEL to write out custom file formats to plug data into a game engine. Custom files can even be used as the backbone for a complete pipeline organization tool.

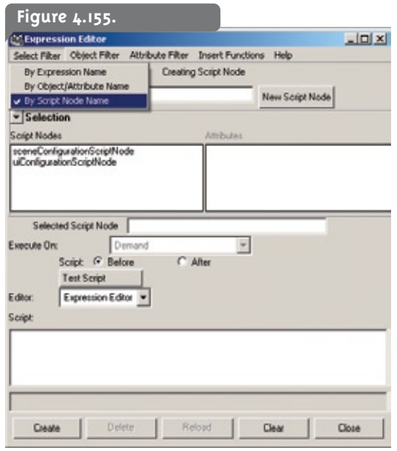
If you want to get some practice with file I/O, try editing this pose file system in some way. One idea that could be useful in a production is to have a pose file that is good for all of the characters in a production. In this system, you could have an extra line for each pose that specifies the character that that pose belongs to. In this way, you could edit the parser to only read in poses that belong to a certain character. There is no end to the level of complexity that you can add to make this system fit your production as needed.

■ Embedding a Script

In our relentless quest to simplify things for the animators, there is one last task. How do we get the animators to actually use the interface? Having your animators search for a script and source it themselves can often lead to disaster. If they are left to search through a complex network of folders to find a script file that they then have to source and call, they likely will not even bother.

Even if your animators are a MEL savvy bunch, searching and sourcing the UI every time they open their scene files is a needless step. Using a 'script node', we can embed the script directly into the scene file. We can even have the interface automatically pop-up every time they open the character file!

Figure 4.155.



To understand how script nodes work, let's go through a mini exercise before I show you how to do it with Zuckafa's UI.

1. Open a new scene file.
2. Open the expression editor (Window>Animation Editors> Expression Editor) FIGURE 4.155 .
3. Under the 'Select Filter' menu, choose 'By Script Node Name'. Now we can create a script node. This is a node that will embed itself in your scene file and execute on a certain condition (this condition is defined soon).
4. Let's write a little command to verify that it is working. How about creating a trusty sphere? Everyone loves spheres. Type 'polySphere;' into the script section of the expression editor.

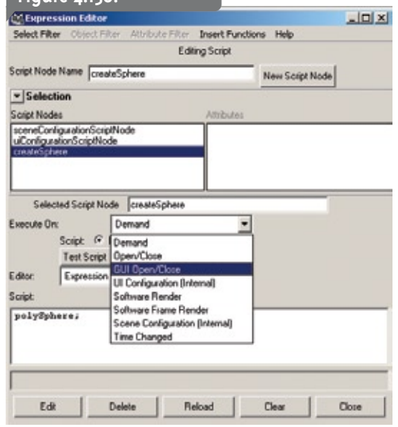
5. Now type 'createSphere' into the 'Script Node Name' text box. It is always good practice to name your expressions. Hit the 'Create' button.

6. We have created a script node. To make this a *useful* script node, we need to define a condition with which it will execute. This is done with the 'Execute On' drop down menu in the expression editor window FIGURE 4.156. Select, 'Open/Close' from the drop down menu.

7. Save the file, name it 'test.mb' or something like that.

8. Re-open the file and viola! You will see a polygon sphere has been created as soon as you opened the scene file. The script node that we embedded in the file was triggered when the file was opened and it executed all of the commands inside it. In this case, there was only one command, the 'polySphere;' command.

Figure 4.156.



To embed Zuckafa's UI into your animators scene files, simply copy the entire contents of the MEL file into the script section of the expression editor and make a script node out of it. To make the interface pop up when the scene is loaded, just add a call to the main procedure at the end of the expression.

Script nodes can be used to embed *any* script you want. They can be really useful for all sorts of things, not just interfaces.

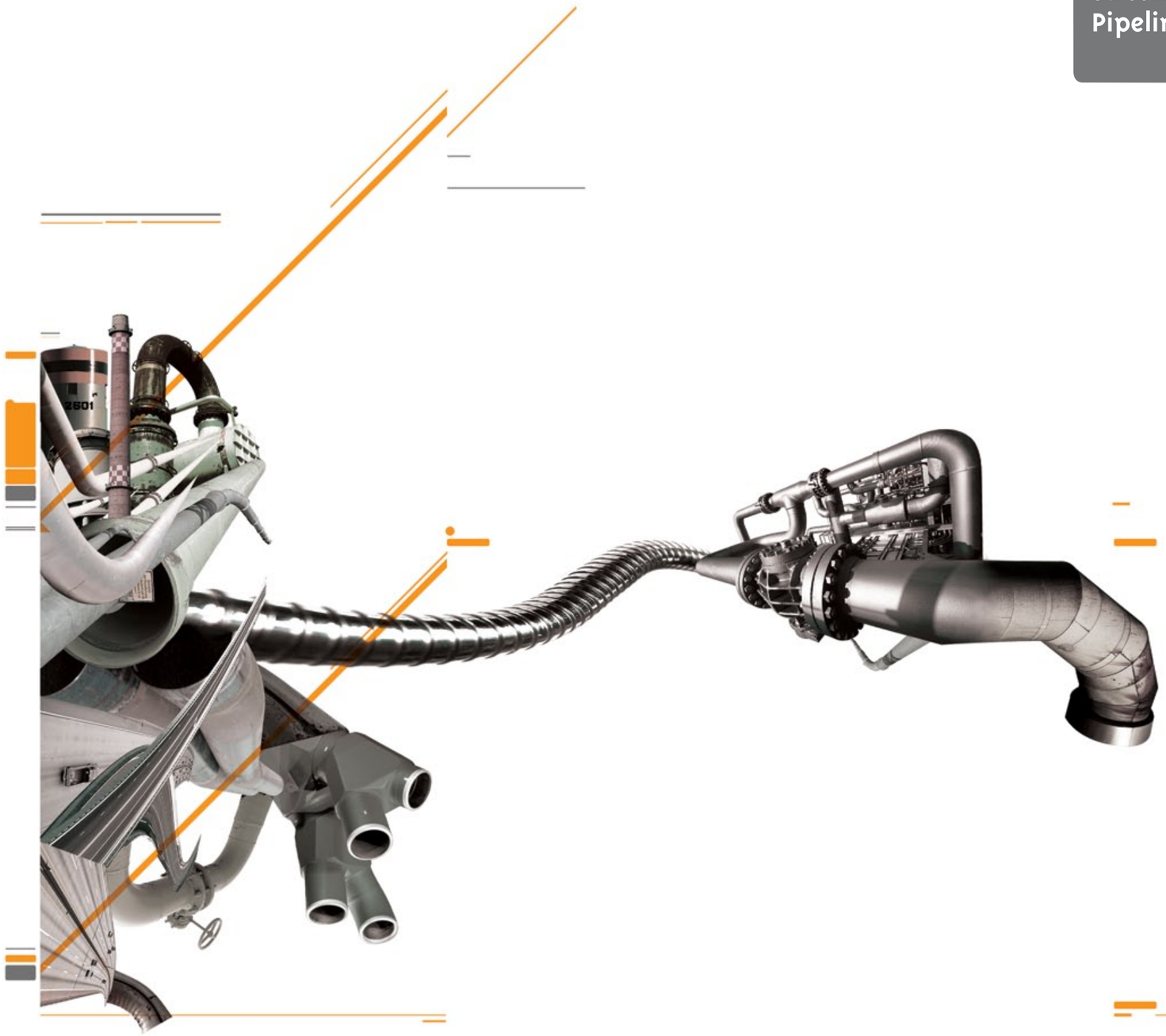
■ **Beyond This Chapter:**

In this chapter, I went into detail to cover the most important aspects of facial rigging. Specifically, I covered:

- How blendshapes work.
- How to model a mesh for animation.
- How to define a shape set for your character.
- How Dr. Paul Ekmans Action Units can help you define a shape set.
- Constructing a shape set.
- How to build and connect an interface for the animator to use.
- How to use Maya's modeling tools to help you sculpt shapes.
- How to use Maya's rigging tools to smooth bind a jaw.
- How to rig realistic, soft eyes.
- File I/O with MEL
- MEL scripting. The creation of Zuckafa's facial UI.
- How to build a program to parse through an ASCII file.
- How to use a script node to embed a script into an .mb file.

With this chapter, I really wanted to impart to you the fundamental concepts needed to construct believable facial animation. There is an old saying that applies here, *"If you give a man a fish, you feed him for a day. Teach him how to fish and you feed him for a lifetime"*. By starting at the beginning, I hope I've taught you how to 'fish' for your own solutions to common rigging problem. You can now use or alter my setup ideas to create the coolest facial rigs ever. Silly fish metaphors aside, I really do feel that you should now have the knowledge needed to make awesome, believable, production ready facial setups.

The next chapter will take the idea of using MEL's file I/O capabilities a step further. In chapter 5, you will be guided through the creation of a production ready script for saving and copying animation data between rigs. This type of script is not only extremely useful, but I am sure you will enjoy this unique guide through the creation of a truly amazing tool.



Chapter 5
Streamlining The Animation Pipeline

A case study in tool development:

In this chapter we are going to cover a lot of ground in MEL. We will be exploring ways of accessing key-frame information, writing to files, dealing with the user interaction, managing data with arrays and even a touch of vector math. We are going to get you from scratch to a finished and polished utility that you can use to help speed up production on your next big game project. Good luck, and remember to relax, this is supposed to be fun!

Once you have gone through the entire chapter and have a pretty decent handle on what is going on in this utility, you will have a tool that can great speed up some of the more boring and tedious parts of computer animation. This will allow you to save animations from characters that you have slaved over for many many hours and apply those animations to other versions of the same character or to completely different characters, regardless of their size. This will also be a great help for copying that "Neutral Pose" that all of the characters' animations start and end with to all scenes with that character in it at once, using the batch functions (especially when that bully of an art director decides that the neutral pose he signed off on a week ago, should be something completely different now). This can be extended to anything with animation information on it, and not just characters. Once you really understand what is going on with the utility, you can tailor it to your needs and really make it an integral part of your character animation pipeline.

In this chapter, we are going to use the Animation Copy and Store utility as a case study on building a complete yet user friendly tool that can greatly enhance the efficiency of your character animation production pipeline.

Specifically, this chapter will cover:

- The creation of the Animation Copy and Store utility in MEL
- Tips on designing an easy to use UI
- Creating Tabs and other UI controls with MEL
- Techniques for handling larger amounts of data
- Techniques for faking multidimensional arrays
- File handling using MEL
- The theory and practice of recursive functions

■ Problem Identification

Writing a script from scratch can often be a very daunting task, especially when you look at all of the features you want your script to have. Things get pretty overwhelming very quickly. Your heart starts to beat out of your chest and you may even panic a bit. In the words of the almighty Douglas Adams, "Don't Panic!". This is good advice. Scripting, or even programming for that matter, is nothing to panic over. Much like starting any kind of project, whether it be starting a new painting, or fixing your house, or even planning your day, the fear of scripting can be overcome by simply breaking everything down into small manageable chunks. *The Animation Copy and Store* utility is no different. Sure it is almost 4,000 lines long, and some of the parts are a bit complex, but really, it all breaks down into small understandable parts. Think of it this way, Maya only knows so many commands. There are a finite number of commands to know. It is actually quite limited. The list gets even smaller when you only consider the commands that are used most often... So, please relax, grab a drink and let's get started.

The first thing that we need to do is list all of the features that we want to have in this utility. Here is the list that I came up with:

We want to be able to store animations to a file.
We want to be able to select a set of objects and store those keys only.
We want to save Poses as well as animations.
We want to be able to store the values as relative values or just keep the joint rotation/translation information. This way we have more flexibility to copy the animation info to many different types of characters. We will probably want to be able to only copy a range of keys instead of the entire animation.

-We must be able to read-in animations and apply them to another rig or object.
-There must be a way to customize what objects the animations are copied onto.
-The script must have the ability to edit the objects that were read in, in such a way that we will not need to copy all of the object's animations into the new file.
-The script must have some features for convenience. If the copying is simple, then we can quickly set it up and get on with whatever we need to do.
-The script must be able to copy animation between rigs of different sizes. This is actually quite tricky, but definetly worth adding.
-We will want to be able to copy the animation to a different frame than the original.

-The script must support batch processing so that we can copy the animation to many files at once.
-We need a way to pick a source file to load.
-It would be nice to have a feature that logs the script's progress.
-We will need a way to allow the user to get a list of files to apply the animations to.

Truthfully, getting that list is one of the hardest parts. The more complete that list is, the easier it will be for you to start implementing the features. It is always easier to drop features than it is to add them in later. So when you are making your feature lists, make them longer than you need. Put all of the features you can think of in there so that you can begin thinking about them while building the major framework of the script. Having a complete list will also help you to determine what bits of the script you can use for many different things, instead of writing the same thing many different times over. If you are just starting out, this will not be immediately apparent, but I promise it will be by the end of all of this.

Basically, the overall purpose of the script is this. We want to be able to copy animations or poses from one character to another, or even to the same character in a different file. We always want it to be fairly straight forward to set up so that the user is not spending tons of time just trying to get his or her animation copied to a different character. Now that you have seen all of the features, take a moment to get all of the panic out of your system and then we can start breaking it down into easy to swallow pieces.

■ User Interface

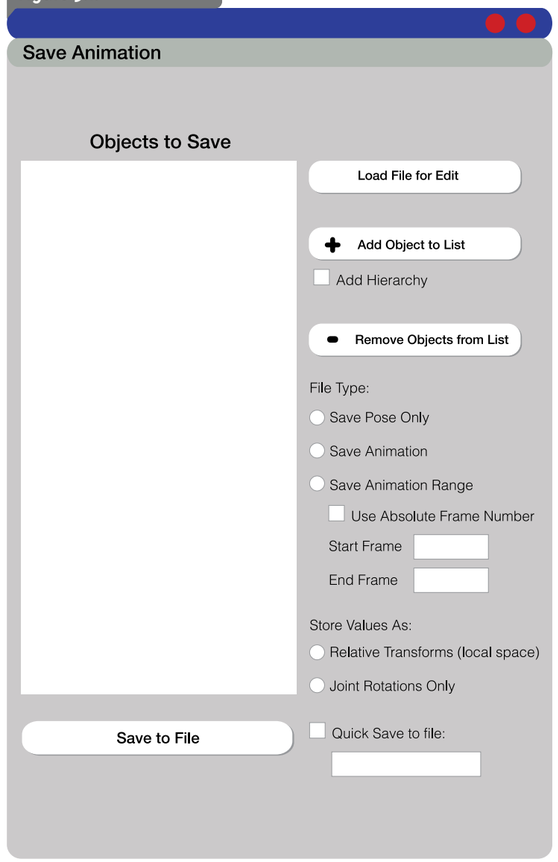
I am the kind of person that likes to get some instant gratification when I am creating things. So after I have a feature list, I start making the user interface or 'UI' (Sometimes called the GUI or graphical user interface). The reason that I do this is because I can quickly get some windows and buttons on the screen that I can play with. We all like to play with the buttons, and just cannot resist pressing them (even the giant red ones that say, "DO NOT PUSH... EVER!"). This provides us with an anchor point for the rest of the script. We can easily test things by hooking the code up to the buttons. Once that is done, we just press them. So for the sake of impatience, and the benefit of a nice testing environment, let's build the interface first.

The first step does not involve any sort of coding or scripting. Just a pencil and paper, or a "paint" program. We want to draw the interface out first. To do that, I took the list of features and broke them up into three parts, "Save", "Load", and "Batch". It seemed that would be a convenient break-up because the three sets of features do not really require anything from each other, as they serve very different functions in this utility. Next, I kicked around the idea of making three separate scripts with three separate interfaces. However, I eventually decided that it would only be a hassle for the user to keep loading up different bits of software to get the job done. So, I looked around at some of my favorite apps and tried to get some ideas on how to "house" a bunch of different features all in one UI.

In the end I decided that 'tabs' were the way to go. That would allow us to keep the features fairly segregated, while unifying the whole interface. Once that decision was made, I went on to designing the UI. Here is the first tab that I came up with: FIGURE 5.1.

This is the 'Save' tab. This is where we include all of the features that involve saving the animation from a character out to a file. One of the main things to keep in mind when designing a UI is to think about information break up and default values. What this means is that features that have multiple choices should be grouped together, broken up by spaces or some other means. For instance, the box on the left is going to be the list of all of the objects the user wants to save the animations for. I wanted to give this the most room in the UI because the user will be spending a good bit of time in that box, editing which objects are going to be stored and which are not. Below that box is the 'Go' button. For scripts like this, I like to have one final button that when pressed sets everything in motion. I like it to be easily recognizable and quick to get to. Therefore, it is on the bottom, below the list, taking up a bunch of room, and not surrounded by a lot of clutter in the UI.

Figure 5.1.



Let's look at the entire right side of the UI. These are all of the functional buttons and the options. The first button is a convenience feature for the user. This will allow the user to load up an animation or pose file that they have saved off with all of the proper objects loaded back into this interface to edit. This will let them add or remove other objects from the file, and then save it back out.

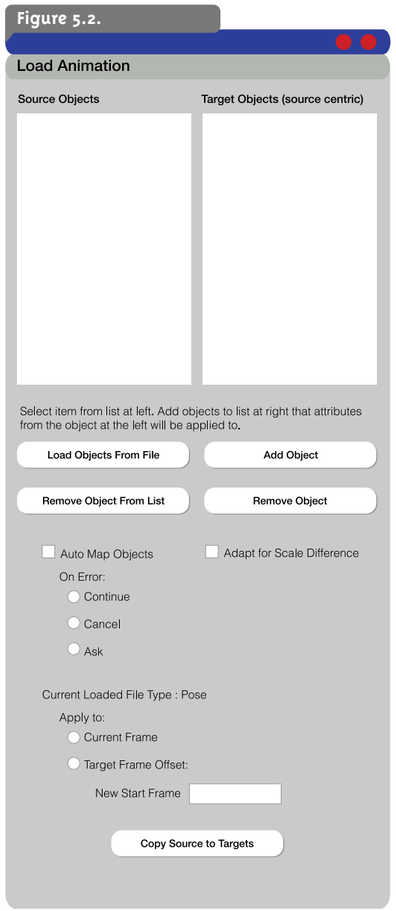
Below that is the add and remove objects from list button. Originally, I had planned on making those two buttons, one button. So if items were selected in the list on the left, they would be removed. If no items were selected in the list, the object selected in Maya would be added to the list. I decided that this would be too confusing if you were in a hurry, and in this industry, we are always in a hurry. So it seemed simply easier for the user if the buttons were split into two. These buttons are most likely going to be used the most, so they are nice and big and not too cluttered up. You will also notice the 'Add Hierarchy' check box. This is another convenience feature, just to help speed up the process of saving your animations off. If this is checked when the "Add object to list" button is pressed, then it selects all of the children of that object and adds them as well. This is nice if your rig is organized that way or if you want to only save off the joint rotations, then this will allow you to select the root joint only and then click the "Add Objects to list" button. The function of the "Remove Objects From List" button is probably pretty obvious. It will remove any selected objects from the list at the left.

Next are the options that determine what kind of file is being saved out. The 'File Type' option will let you save the pose only, (meaning the current frame only), the entire animation, or some subset of the entire animation. The 'Use Absolute Frame Number' option will allow you to save the animation as if the subset you have set started at zero, or save the actual 'absolute' frame number that the animation really sits at on the timeline. The usefulness of this option will become obvious later.

Below that are the 'Store Values As' options. One being Relative Transforms and the other being 'Joint Rotations' only. This means that you can either save the relative values that show up in the channel box when you select an object, or you can load a set of joints and save the rotation and scale values of the joints. This will also save the translation values of the root joint.

Lastly is the 'Quick Save to file' option. If that is checked, then you can simply type in the path to the animation file that you would like to save and click 'Save to File'. This will skip the file save dialog box that usually pops-up and just save the file out, no questions asked. This is a good feature for power users. Especially if you are tweaking a larger animation file.

Figure 5.2.



The design above is the 'Load Tab'. FIGURE 5.2. This is the tab that will allow the user to load in a previously saved animation file and then apply it to the current scene. This consists of four sections. The first one contains the buttons that allow you to customize the 'mapping' of the animation. In other words, if you have a file loaded that has objects with different names in it than the animation file you saved out, this will allow you to assign targets to the objects found in the animation file. This is very handy, especially for copying animations from one character to another. The next section contains options on how to apply the animation. Section three has file type specific options on where on the timeline to copy the animation to. Finally, we have the 'Go' button again. This is what the user will press to actually copy the animation to the new objects.

Section one consists of two lists and two buttons for each list. The list on the left is the 'Source Objects' list. This will hold all of the names of the objects stored in the animation file saved from the 'Save' tab. The list at the right will hold the names of the 'targets' that will receive the animation information from the 'source' objects.

The two buttons on the left under the 'Source Objects' list will allow the user to load an animation file into the source objects list and into memory. The 'Remove Object From List' button will allow the user to remove an object from the source list. This comes in handy if the character you are copying the animation to does not have all of the same features as the source character.

The other two buttons under the 'Target Objects' list will allow the user to add custom targets to the object selected in the 'Source Objects' list.

Section two will tell the utility which objects to copy the animations to and whether or not to adjust the animation based on the difference in scale between the source object and the target object. If 'Auto Map Objects' is checked, then it will simply find the object in the current scene with the same name as each object in the 'Source Objects' list and copy the animation to the object with the same name as the source. If this is checked, the user will also be able to tell the script what to do in case of an error. The user can choose between Continue, Cancel, or Ask. This means, that in the case of an error, the script will either ignore it and move on to the next object, stop the script completely, or ask the user what they want to do.

The third section allows the user to either copy the animation to the same frame numbers that are stored in the animation file or to copy them using an offset. Remember the 'Absolute Frame Number' option from the 'Save' tab? This is where that comes into play. The 'Current Frame' option will read the frame numbers stored in the animation file. So if you checked the 'Use Absolute Frame Number' box, then it will apply the animation using those saved frames. If you did not use that feature, then it will start applying at frame 0. This is of course assuming that you did not select 'Target Frame Offset'. in the 'Load' tab. If so, then it will add to the saved frame number, the number the user puts in the 'New Start Frame' field.

Lastly, we have the 'Copy Source to Targets' button. This makes it all happen. Once pressed, it will read in all of the options and start to apply the animation.

Figure 5.3.

The last tab is the 'Batch' tab. FIGURE 5.3. This is the tab that will allow the user to apply the same animation or pose to many different scene files. This is broken up into two halves, much like the 'Save' tab. If you will notice, the layout of the 'Batch' tab is pretty much a mirror image of the 'Save' tab. This was a decision, not an accident. I did not want the two tabs to have the 'Go' button in the same place, simply because I did not want a user that has been working on a project for the last 90+ hours with no sleep, no coffee (because the coffee maker broke), and no patience left, to accidentally think he was on the 'Save' tab instead of the 'Batch' tab. We've all been there, and we all know how simple mistakes like that can be made, and make you want to quit being an animator. So, while designing your interfaces, please keep things like that in mind. Just remember all of the things that you did not like in other software packages, and simply find a better way.

As mentioned earlier, this is broken up into two halves. One half being all of the options for the batch job, and the other being the list of files that the animation or the pose is going to be applied to. On the left we have a few items. The first being the 'Source File'. This field will contain the name of the source animation file to load up. The button will bring up a file browser dialog. Next is the 'Log File'. This is the name and path to the file that all of the errors or messages from the script will be sent. The button below that simply brings up a file browser dialog. Below that is the 'List Type' option. This will determine if the list contains the names of files or the names of folders that contain the files. If the 'folders' option is selected then you will also be able to check the 'Recursive' box to tell the script to look into all of the folders within the loaded folder. Next-up are the logging options. Logging style tells the script what types of things to log, whether that be errors only, or every message that is written out to the script editor. The other logging option is the 'Log File Setting' option. This will tell the script to either append the new log to the end of a file that already exists or simply overwrite it with the newly logged items. The last two buttons add files or folders to the list, or removes them. If you have 'Recursive' checked before you add an item to the list, then that folder is marked with an asterisk (*) next to it, to let the user know that the script is going to recursively search that folder for files.

That about does it for the interface design. Just remember, always keep in mind how the options and information are broken up, and which ones are used the most. The last thing to remember is what default settings should be used. These should be set to be the most convenient for the most people. Remember, you want to make the users' lives as easy and efficient as you can, because chances are, if you are writing a script, you are going to be one of those users.

■ Making A Window

Whew! We just covered a bunch of things in a very short amount of time. There is so much to be said about interface design that a great many books have been written on the theories of that subject alone. Take a trip to your local library or find some texts at a nearby bookstore to try and get a feeling for exactly what interface design is all about. The interface is basically the only thing that the end user is going to care about, so make it good!

Now is probably a good time to take a break, walk around and stretch, and maybe grab a drink. The next part is taking all of these plans, and building the actual interface in Maya!

Before you get settled back in your chair, you are going to need a few things. Make sure that you have a copy of Maya installed (obviously), the Maya Help open, and a text editor. Chapter three has some suggestion on text editors, or you can simply use notepad if you like. However, a real programmer's editor will make your life much easier. I mostly use Emacs if I am on a Linux box, or Lugaru Software's Epsilon, when I am on a windows system. UltraEdit and MED are both good editors as well. There is also a Maya plug in called 'MEL Studio'. Mel Studio is a MEL-specific editor that is completely integrated within Maya. This is also an excellent choice.

Remember how I said that there are a finite number of commands in MEL? Well, I wasn't lying. Please open the Maya help if you have not already done so and go to the Developer Resources Section. In that section there is a 'MEL Commands' link. Go ahead and click that link. This is a list of everything that MEL can do. No kidding, it is all there. This is a great resource, please keep it open at all times, as it is the best place to read up on commands. With this list at the end of your "F1" key, there is no need to try to memorize all of these commands. If there is any time that I mention a command that you are unsure of, please look it up here, as it is all there for you in one place. OK, let's get started!

Fire-up Maya and set yourself up with a new project folder (you do, do this for every project don't you?). Now open up your text editor and save a file into the MEL folder in the project you just set up. As you work, please remember to *save early and save often*. Also, make as many backups as you think you need. Don't feel bad about making hundreds of versions. Who cares if your hard drive is messed up a bit?

The first thing that you want to put at the top of each of your scripts is a small block of comments that describe the script. This will help you to later remember what the script does, and help others to use your script. Here is what I put at the top of the jcAnimCS.mel script:

```
//              Animation Copy and Store
//      Copyright © 2005 CGTOOLKIT. All rights reserved
//                  www.cgtoolkit.com

// Script: jcAnimCS ()

/*
        General Description:
        -This script will facilitate the common task of saving
         animation from a creature rig.
        -Users can copy animation between rigs regardless of any
         differences in scale.

        How To Use:
        1.) Copy jsAnimCS.mel from the DVD to your script folder
        2.) Type jsAnimCs; into the command line.
        3.) Press 'Enter' to bring up the script GUI.

        Author: Jake Callery
*/

//The rest of the script goes here...
```

Fist off, let me explain a little about comments, as they are one of the most important parts of any script. These are little notes that you write for your self. Code can be tough to read sometimes. There are things that you can do while writing code to make it easier for you to read later. The biggest thing is comments. Since these are just little notes, they are not actually read by Maya, they are simply ignored. This means that you can write anything you want in a comment.

Now there are two ways to write comments in MEL. The first way is to simply put two forward slashes in front of the comment. Like this:

```
//This is a comment! I can write anything I want after the two
//slashes, but never say anything rude!
```

Once you put the two forward slashes in, anything after that is ignored by Maya. The other way method can be seen above. This method will allow you write entire blocks of comments. We will use this at the top of each script and before each procedure or function to explain what the script or procedure does. This way we do not have to read the code to figure it out, just read the comments. The second way of writing comments goes like this:

```
/* This kind of comment can be on multiple lines,
unlike the other style of commenting that is for only one line.
*/
```

To start the comment, you put a forward slash followed by an asterisk. Then you write whatever you want for the comment. When you are

finished writing the comment, you simply put an asterisk followed by a forward slash. This method is nice for doing comments that take up multiple lines.

Now that you understand comment blocks, write a block of comments that explain what the script is supposed to do, just like I have done above. Most of the items are self explanatory, however the first line is a bit strange:

```
Script: jcAnimCS()
```

This is basically stating what needs to be called from Maya to run this script. This is the main 'Global Proc' for the script. We will cover that soon, when we create a window to work in.

As I just stated, the first requirement for an interface is a window to hold the interface. So let's build a window!

The first thing that we need to do is create a global procedure for Maya to call. Basically what this means is that we are handing Maya the name of our script, so that the user can type in that name and run the main part of our Script. This is what a global procedure looks like:

```
/******** Global Functions *********
***********************************
jcAnimCS()

Arguments:
        none

Returns:
none

Notes:
The main global proc Name needs to be the same as the file name
(minus the .mel of course)

*****************************************************************
*******\
//The main global proc Name needs to be the same as the file name
//(minus the //.mel of course)

global proc jcAnimCS()
{//jcAnimCS
        //The body of my procedure goes here
}//jcAnimCS
```

After the main comment block at the top of the script, add this code. This is another comment block before the global proc. It is good practice to make one of these for each of your procedures. Add the name, what arguments it takes, and what it returns. We will get into what that all means a little later on. The next line after the comments is:

```
global proc jcAnimCS()
```

this is how you declare a global procedure. The 'global' part tells Maya that this procedure is available to all scripts to use, and for all of Maya to use. 'Proc' means that it is a procedure that we are writing, and that is followed by the name of the procedure and then an open and close parenthesis, which is where the arguments are defined, if we had any. Please note, that the name of one of the global procs in your script must be the same as the name of the .mel file itself. So since this proc is named jcAnimCS, the MEL file needs to be called jcAnimCS.mel.
The only thing left to complete this procedure is the open and close brackets "{" and "}". These define the start and end of a code block. Basically, in this case, it means that the procedure actually starts at the "{" and then ends at the "}". Notice that after each of the brackets we put the name of the procedure in comments. This is to help us keep track of what procedure we are working in.

Now that the frame of the procedure has been written, let's put some code in it to make sure that everything is working properly. Where I have the 'The body of my procedure goes here' comment, insert the following line of code:

```
print ("MEL is great!\n");
```

When this is executed, the script editor should display:

```
MEL is great!
```

I did not mention what the 'print' command actually does, so let's look into that. Go into the MEL commands section of the Maya help and look it up. This is what you will find:
"The print command displays it's argument in the Script Editor (or in standard output in batch mode). If the argument is not a string it is converted to one."

Now before I explain what all of this means, save the script and let's run it for the first time! After you have saved the file, go into Maya and open up the script editor. Under the 'File' menu, you will find 'Source Script...'. Select that, now point it to where you have saved your .mel script, and select 'Open'. What this does is load the new script into memory. Now Maya has access to the script file's contents. Please note that every time you change the script file, it needs to be 'sourced' again. Maya will not pick up the changes to the file, because it reads the script from memory and not from disk. This is the first chance that Maya has to look at the script and let you know if there is anything wrong with it. If you do get an error message, please make sure that your script looks like this one below:

```
//              Animation Copy and Store
//      Copyright © 2005 CGTOOLKIT. All rights reserved
//                www.cgtoolkit.com

// Script: jcAnimCS ()

/*
        General Description:
        -This script will facilitate the common task of saving
         animation from a creature rig.
        -Users can copy animation between rigs regardless of any
         differences in scale.

        How To Use:
        1.) Copy jsAnimCS.mel from the DVD to your script folder
        2.) Type jsAnimCs; into the command line.
        3.) Press 'Enter' to bring up the script GUI.

        Author: Jake Callery
*/

//The rest of the script goes here...
```



```
/******** Global Functions *********
***********************************
jcAnimCS()
Arguments:
        none
Returns:
none

Notes:


***********************************\
//The main global proc Name needs to be the same as the file name
//(minus the //.mel of course)
global proc jcAnimCS()
{//jcAnimCS
        print ("MEL is great!\n");
}//jcAnimCS
```

Once you source the file and it passes without errors, go down to the input area of the script editor and type:

```
jcAnimCS();
```

Now press CTRL-Enter.

"MEL is great!", should have been printed into the history part of the script editor. If this is the case, then congratulations! You have written a successful MEL script!

There are a couple of things I sort of glazed over about the *print* line. So let's go over that. The *print* procedure is much like any other procedure that you would write. You call it by its name and you give it (pass it) arguments. Arguments are simply values or information that you provide the procedure that it needs to complete it's task. In this case we are giving the *print* command a line of text to print in the history part of the script editor.

There are different types of information that we can pass to a procedure. Here is a list of some of them:
int - a whole number, no decimal point (15)
float - a number with a decimal point (15.345)
string - characters or text, usually found between double quotes ("This is a string")

I simply wanted to introduce these here. We will get much deeper into these *data types* later in the chapter. For now let's simply concentrate on the *string type*.

As you can see above, we are passing the *print* command a string, since it is characters or numbers surrounded by double quotes, like this, "MEL is great!\n". The "\n" part is a little bit tricky. This is called an escape sequence. The backslash is an escape character and it will sit in front of another character. In this case, "\n". The "\n" escape sequence will create a new line break. So if we had two print commands in a row with no "\n", then those two strings would be printed on the same line. Give it a shot by duplicating the 'print' line in the script and then remove the "\n" characters. That will print:

```
MEL is great!MEL is great!
```

Iff you put the "\n" back in, then it would read:

```
MEL is great!
```

The last thing about the *print* line is the semi-colon ";". This is known as an end terminator. It specifies where you are done with that line of code. Without an end terminator, Maya will continue reading the next line as though it belonged to the previous one.

There is one more thing that I need to briefly explain, variables. Variables are simply containers for data. They can be several different types (int, float, string, matrix, or vector). This means that you can use a variable to hold one item (with the exception of arrays, which we will get into later). All variables start with the dollar sign "$". This special character tell's Maya that the next word is the name of a variable. A quick example can show this best:

```
int $i;//Declare the variable, so Maya knows what type of data
//its going to hold
$i = 10;//Assign a value to the variable
print ($i);//Print out that value
```

The first line tells Maya what type of Data the variable '$i' will hold. In this case, it can hold an integer, which is a whole number. Next we assign a value to the variable. This essentially stores the "10" into $i. So now, when we print out $i, "10" will be displayed in the script editor.

With all that variable stuff fresh in your mind, let's get on with displaying a window. Go back to the text editor, and remove the *print* lines. Put this in it's place.

```
//Create Main window (jcAnimCSWin)
string $myWindow;

$myWindow = `window
                        -title "Animation Copy & Store"
                        -minimizeButton on
                        -maximizeButton off
                        -titleBar on
                        -sizeable on
                        jcAnimCSWin`;
//The above command will create a window and store it into
//Maya's memory. The flags specify attributes on the window,
//Now that the window is in memory, we can display it...
showWindow $myWindow;
```

Now, save-out the script again, source it, and run it. You should have a nifty little window within Maya! FIGURE 5.4.

Figure 5.4.

Let's discuss what is going on here, shall we? The first statement after the comments is a variable declaration, just as we have seen before.

```
string $myWindow;
```

This simply creates a variable that can hold strings. Next, a value is assigned to the variable with the '=' operator:

```
$myWindow = `window
                        -title "Animation Copy & Store"
                        -minimizeButton on
                        -maximizeButton off
                        -titleBar on
                        -sizeable on
                        jcAnimCSWin`;
```

The part of this statement that we have not yet gone over, is the 'back quote', or 'back tick'. This is the, ` ` ` character. Surrounding a statement in back quotes will execute that MEL command first and then substitute the returned value in it's place. Many commands return useful information, back ticks can capture this in a convenient way.

So what this is doing is executing the *window* command. Have a look at the MEL help and look up window. This will tell you all about it. So you can see that we have a command with a set of flags and lastly the name of the object to use the command on. Let's go through the flags first:

-title : This specifies the title of the window.
-minimizeButton on : Enables the minimize button for the window.
-maximizeButton off : This disables the maximize button for the new window.
-titleBar on : This enables the titleBar.
-sizeable on : This lets the user resize the window.

Finally, there is the name of the window:
jcAanimCSWin - This tells Maya what name we want to refer to the window with. This allows us to query the window.

In the help, you will see that *window* has a return value. The help states that *window* will return the, "Name to the window". This means that once the window command is executed, it will store "jcAnimCSWin" in the $myWindow variable. Now the $myWindow variable can be used to refer to our window.

Alone, the window command does not actually show us anything. We need to call the showWindow command to display the window from memory.

```
//Now that the window is in memory, it's time to display it
showWindow $myWindow
```

That command will actually put the window on the screen.

Now run the script again. You may get an error like this:

```
// Error: line #: Object's name is not unique: jcAnimCSWin//
```

Maya cannot keep track of two windows that have the same name. This means that we have to close any of the windows of that name before opening a new one. However, we do not want to rely on the user to have to worry about this, so we will make Maya worry about it. Add the following code after the $myWindow variable declaration:

```
if(`window -query -exists jcAnimCSWin` == 1)
{ //Window Exists
  //The window already exists, so delete it before
  //making the next one
  print "Window Exists! Deleting it to create the new one...\n";

  deleteUI jcAnimCSWin;
}//Window Exists
```

The first statement after the comments is something called an '*if*' statement, or *conditional*. This is designed to compare one value to another. They can be, equal to, not equal to, greater than, or less than. Once the comparison is made, it will be evaluated as true or false. If it is true, Maya will execute the commands between the brackets. If the comparison is false, then Maya will resume execution of the script after the final closing bracket for the "if" statement. For example:

```
//This would be true
if(5 < 10)

//This would be false
if(10 == 13)

//This would be true
if("string1" != "string2")
```

The item in the "if" statement that actually does the compare is called a '*conditional operator*'. These are the common operators:

== "Equal To": Is the value on the left, the same as the value on the right? Please note that this is different than just a single "=", which assigns values to variables and does not compare them.

!= "Not equal to": Is the value on the left not the same as the value on the right?

< "Less than": Is the value on the left less than the value on the right?

> "Greater than": Is the value on the left greater than the value on the right?

>= "Greater than or equal to": Is the value on the left greater than or equal to the value on the right?

<= "Less than or equal to": Is the value on the left greater than or equal to the value on the right?

As we go though this script we will go over these time and time again. So if you are a little confused now, its OK, it should clear up for you with time. They become second nature with practice.

So back to the task at hand. We have learnt that the use of the back quote executes the commands that it surrounds. So `*window -query -exists jcAnimCSWin* `, is executed first thing. This will either return a 1 or a 0 depending on if the jcAnimCSWin window exists or not. Here is how this works:

We execute the window command again, this time with a different set of flags. This time we are using '-query' and '-exists'. The query flag sets the command in query mode, which means we can ask Maya a question about something, and the -exists flag, is that question. This says, does the jcAnimCSWin window exist? If it does, it will return a 1 and if not, it will return a 0. Then the 'if' statement compares that value to the 1. If the window does exist, the 'window' command will return 1 and that will be compared to 1. If that happens, the 'if' statement is true and the code between the braces is executed. If it is 0, then the 'if' statement is false and the code is skipped.

Assuming that the if statement is true, we print out a message to the user in the script editor, that says the window already exists, then we delete that window using the *deleteUI* command. The only argument that the deleteUI command takes is the name of the window. This will delete the window so that we can make a new one.

We now have a fully functional window to start putting the UI in! Now would be a good time to take a little break, and maybe play around with the print command, the 'if' statements, and the window command a bit. Once you feel a little more comfortable with all of this, come back and we will move on to adding UI elements.

■ Building the UI

Constructing a UI in MEL is much like making HTML tables. The code looks completely different mind you, however, the concept is the same. You start with a big table, then you can fit smaller tables inside of it. In Maya, these are called layouts. Each type of layout is slightly different in what they can contain. Some of the ones that we will be using are the tabLayout, columnLayout, and rowLayout. These are the most common layouts and are used in almost every interface.

The way this chapter is written is going to change slightly from here on out. Since the script is so big, I am just going to work though the entire thing, line by line, instead of "building" it back up before your eyes. Please also keep in mind these two things. First off, the Maya help is always there for you and is probably the greatest resource to you, and two, "DON'T PANIC!" . Please open up the jcAnimCS.mel script file in your editor and locate the //***Tab Layouts*** line. This is where the UI starts (notice it is in the jcAnimCS global proc, which is the same name as the script file).

Earlier, we discussed that the utility is broken up into three sections; Save, Load and Batch. The tabLayouts are going to create the tabs for us. As I mentioned, layouts are containers for buttons, fields, and even other layouts. So this is what we have:

```
//***Tab Layouts***
tabLayout;
```

The tabLayout will not show in the UI until it has a child control placed in it. Once there is a child, the tab will be shown and it will show the name of the child control in the tab text itself. So let's add the child control:

```
//***** Save Tab *****
//main rowLayout for the "Save" tab
//this tab is split up into 2 columns that hold
//the lists and the buttons
rowLayout
        -numberOfColumns 2
        -columnWidth2 200 200
        -columnAttach 1 "both" 5
        -columnAttach 2 "both" 5
        save;
```

What this does is start a rowLayout that contains two columns, each having a width of 200. The columnAttach flags attach the edge of the column horizontally, and then creates a five pixel buffer between the columns. Finally, the last argument for the rowLayout control is 'save'. This is the name of the rowLayout. We will be able to identify the layout by this name. Also, as it's the only child layout of the tabLayout, the tabLayout displays and shows the 'save' string in the tab title.
Now, remember how I mentioned that a layout can contain other layouts? This is what we are going to do next. Right now, as it stands, we have a tab that holds a rowLayout with two columns. This means that we now have two columns to put other controls in. It works like this; the first item that you place in a rowLayout will get shoved into the first column. The next item will get shoved into the second column. Once that happens, all of our 'slots' are filled up in the rowLayout. Since we have more than two items that we want to add to the 'Save' tab, we need to find a way to add more slots. To do that, we add yet another layout inside the rowLayout. This time, we are going to use a columnLayout. A columnLayout works like this; each item that you add into the layout simply gets shoved below the previous one. This means that we can place as many controls as we want into a columnLayout and they will simply stack on top of each other. This is great for us since the "Save" tab is basically split into two columns. So let's chuck-in a few pieces of the UI. Please remember to comment your code, so that future readings will still make sense to you!

```
//Column 1
columnLayout
        -adjustableColumn true
        -columnAlign "center"
        saveColumn1;

//Label for the text list
text
        -label "Objects To Save"
        -height 25
        -align "center";

//List on the left
textScrollList
        -numberOfRows 31
        -allowMultiSelection true
        saveObjectTextList;

//spacer
//this is just used to make some blank space to help the UI
//look a bit less overwhelming
text
        -label ""
        -height 10;

//Save to file Button on left
button
        -label "Save To File"
        -command "$filePath = `textField -query -text
        quickSaveText`;\jcSaveAnimFile(\"saveObjectTextList\",\
        $filePath,\
```

```
`radioCollection -query -select saveTypeRadioCol`,\

`intField -query -value saveStartFrameField`,\

`intField -query -value saveEndFrameField`,\

`checkBox -query -value absFrameNumberCheck`,\

`radioCollection -query -select valueTypeCol`);"

//Finally, the button command is finished.
saveToFileButton;
```

This starts our first column, which is the left most column. The first part, we have already touched on. The columnLayout command starts a new columnLayout for us. This means that we can start stacking things into that column. We want to set the -adjustableColumn to true and the -columnAlign to "center". This will make sure that the columnLayout fills out as much space as possible in it's parent control. So, basically, this will stretch everything in the column to fit the width of the rowLayout which is it's parent or container. The columnAlign flag tells Maya how to align the buttons, text, and various other things that will be placed in the column. For this one, we will choose 'center'. The last argument for the columnLayout is the name of the layout. This is not strictly necessary, but it is a good idea to name everything that you can so that you can easily access it later in the code. This one we are going to call "saveColumn1".

Before we move on, this might be a good time to talk a little bit about white space. White space is simply the spaces between words in the code. There are times when white space matters and times when it does not. While working through this script, please keep an eye on how white space is used. In the columnLayout, you will notice that all of the flags (options signified by a '-') are on their own line. Maya is fine with this, as well as fine with putting everything on one line. The only thing Maya really cares about is where that semi-colon is. This, as mentioned before, signifies the end of the statement. So, after the name of the columnLayout, you will find the semi-colon. This tells Maya that we are done with this command, and it can go ahead and process it. Once its done with that, it will move on to the next command. The main reason that I split the flags up onto their own lines is so that its easier to read.

Next up is a control that we have not yet discussed. This is the 'text' control. This is essentially a bit of text that you can use to label buttons, or whatever you want to do with it. In this case I am using it as a label for the textScrollList that comes up next. The flags for the 'text' control are fairly straight forward. There is the -label flag, which sets the text you want to display in the UI. The -columnAlign flag is one that we have gone over already, which simply tells Maya where in the column to place the text. In this, case we are centering the text. Lastly, we see the -height flag which tells Maya how tall to make the slot that holds the text. Please note that this flag does not actually change the size of the text, it simply adds the needed space above and below the text. There is also a -width flag which does the same thing, only horizontally.

Let's look at the big one. This is the textScrollList control. This will allow you to create boxes that will hold a list of items, which is perfect for holding a list of objects that we want to save the keyFrame information for. The flags work as follows; the -numberOfRows flag sets the number of items that can be stored in the list before a vertical scrollbar appears. Please note that this does not limit the number of items that you can have in the list, it simply sets it's height. As previously stated, if there are more items in the list than rows, Maya simply puts a vertical scrollbar that will let the user scroll through the extra items. The second flag is the -allowMultiSelection flag. When this is set to 'true', the user is able to select more that one item by either holding down a modifier key (`ALT`, `CTRL`, `SHIFT`) to select multiple items, or by simply dragging across the items. Lastly, there is, of course, the name of the control, which I have made 'saveObjectTextList'. As you are writing your own scripts, please try to help yourself out by coming up with a naming convention that you can call all of your controls by. For me, I decided to start everything with the name of the tab that it was in and what kind of control it is. So in the name, 'saveObjectTextList', the 'TextList' part is the type of control. Please note that simply adding a textScrollList, does not actually get us anything beyond displaying a textScrollList. If it did, where would the fun be in scripting?

The next item is our friend the 'text' control. However, this time, we are not going to use it to display text, but to simply put a spacer in to spread the interface out a bit. So we set the label to be blank, by simply putting in a set of empty double quotes (""), and then set the height to our desired spacer height. In this instance, I set it to ten pixels.

Lastly, in this column, is the 'Go' button that we spoke of earlier. Creating a button is rather simple, and is much like creating anything else in a UI. We simply call the button command, feed it some flags, and then give it a name. The first flag should be pretty familiar to you by now. It is the -label flag, which works just like it does for the 'text' command. When you set it, it will place the label text on top of the button. The next flag is probably the most important one for a button. This is the -command flag. For now, ignore everything between the double quotes as we will return to that later. What the command flag does is allow you to execute one or more commands when the button is pressed. All of that mess that is between the double quotes, are the commands that will be executed when the 'Save To File' button is pressed. The last part is, of course, the name. Following the naming convention I settled on for this UI, I chose to call it 'saveToFileButton'.

We have reached the end of what he have planned for that column. So how do we start working on the second, right-most column? Remember that we are currently adding things to the columnLayout entitled 'saveColumn1'. The columnLayout's parent or container is the rowLayout entitled 'save'. So what we need to do is instruct Maya to add the next item to the rowLayout which will be the second column. Recall that the first item is the rowLayout in the first column, and the second item is placed into the second column. In this case, the columnLayout, 'saveColumn1', fills-in the first column of the rowLayout. We need to add a second item. To tell Maya where to put the next item, we call the setParent command:

```
//Back to main "Save" rowLayout
setParent save;
```

This tells Maya to go back to the *save* layout and start adding the next items there. You will also see setParent used as follows:

```
//Go up to the first immediate parent
setParent..;
```

This, like the comment states, will instruct Maya to place the next item starting at the parent of the current control. If this is not fully clear right now, do not worry, you will get the hang of it when we come across it in the code again.

Now that we have instructed Maya to start at the *Save* control, we must start adding our second column:

```
//Column 2
columnLayout
        -adjustableColumn true
        saveColumn2;

//spacer
text
        -label ""
        -height 25;

//Load file for Edit button
button
        -label "Load File For Edit"
        -command "jcLoadAnimFile(\"saveObjectTextList\",\"\");"
        loadFileForEditButton;

//Spacer
text
        -label ""
        -height 37;

//Add object button
button
        -label "<< Add Object To List"
        -command "if(`checkBox -query -value addHiCheck`){select
        -hi;}
        \$myNewObject = `ls -sl`;\
        jcAddToList($myNewObject, \"saveObjectTextList\", 0);"
        addObjectToSaveListButton;


//Spacer
text
        -label ""
        -height 5;
```

Much like the first column, we start the second one with a columnLayout so that we can add a bunch of items that are stacked on top of each other. This one also has adjustable columns and is entitled saveColumn2. If we now put our first control in, which is the 'Load File For Edit' button, it would start at the very top and be aligned next to the 'Objects To Save' text control. It would be fine there, however it would not look very nice. So we add a spacer first of the proper height to line the button up with the top of the textScrollList. Then we add the button, much like we did before. It has a label, which is a string because it's between double quotes (""), a command flag, and a name. The -label flag takes a string type, which is an important thing to know. This means that we could put a variable name there like $myString that holds a string. So we could do something like the following:

```
//Declare the variable
string $myString;

//Assign a value to the variable
$myString = "<< Add Object To List";
```

This will stick the '`<< Add Object To List`' string into the $myString variable. We can now use the $myString variable instead of putting the string directly in to the label like so:

```
//Add object button

button
-label $myString
-command "if(`checkBox -query -value addHiCheck`){select -hi;}\
$myNewObject = `ls -sl`;\
jcAddToList($myNewObject, \"saveObjectTextList\", 0);"
addObjectToSaveListButton;
// Using $myString instead of ""
```

This would produce the same result. Again, ignore what is in-between the double quotes for the -command flag, as we will get back to 'hooking-up' the interface to actual code later. After the addObjectToSaveListButton, we have another spacer and then the following code:

```
//Layout for checkBox
rowLayout
        -numberOfColumns 1
        -columnAlign1 "left";

//Add Hierarchy
checkBox
        -label "Add Hierarchy"
        addHiCheck;

//Back to main column layout
setParent..;

//Spacer
text
        -label ""
        -height 10;

//Remove Objects button
button
        -label "Remove Object From List >>"
        -command "jcRemoveFromList(\"saveObjectTextList\");"
        removeObjectFromSaveListButton;

//Spacer
text
        -label ""
        -height 37;
```

What we want to do is add the 'Add Hierarchy' checkbox below the 'Add Object' button. In order to do that, we call the checkBox command, and give it a label and a name, much like everything else. However, if we simply put that in as the next item, the checkbox and the text would be centered in the column. This happens because we told the rowLayout to center everything in the column (*-columnAlign "center"*). This will work just fine, however it will not look as nice as if we were to left justify the check box under the 'Add Objects' button. So, to make this happen, we need to give the checkBox it's very own little layout where we can set different alignment rules. We can do this by adding a rowLayout with the -numberOfColumns flag set to 1. This tells Maya that we want to add a rowLayout with one item in it. Then we set the -columnAlign flag

to 'left', which forces everything in that rowLayout to be left justified, which is exactly what we are looking for. Now that that rowLayout is set up, we can add the checkBox below it. Once that is done, all is well and good until we try to add the next control. Since we set this rowLayout to only have one column, we cannot fit anymore into it. We need to tell Maya to add the next item into the 'saveColumn2' columnLayout. Since this is the direct parent of the rowLayout that the checkBox is in, we can simply call 'setParent..' which will put us back into the 'saveColumn2' layout. With the heirarchy set back, can now add the next control.

Now that we are back into the correct layout, we add a spacer and the next button. This is the 'removeObjectFromSaveListButton' button. After that, we add another spacer to help structure the look of the UI a bit more. Then we need to add some radio buttons:

```
//Layout to hold the radiobuttons
columnLayout;

//Text title
text
        -label "File Type:";


radioCollection saveTypeRadioCol;

radioButton
  -label "Save Pose Only"
  -onCommand "intField -edit -enable false saveStartFrameField;\
              intField -edit -enable false saveEndFrameField;"
  savePoseOnly;

radioButton
        -label "Save Animation"
        saveAnimation;

radioButton
  -label "Save Animation Range"

  -onCommand "intField -edit -enable true saveStartFrameField;\
              intField -edit -enable true saveEndFrameField;\
              checkBox -edit -enable true absFrameNumberCheck;"

 -offCommand "intField -edit -enable false saveStartFrameField;\
              intField -edit -enable false saveEndFrameField;\
              checkBox -edit -enable false absFrameNumberCheck;"

  saveAnimationRange;
```

If you look at the original design for the 'save' tab, you will see that we need three radio buttons to set our file saving options. Radio buttons are a bit different than the rest of the controls. Radio buttons allow the user to pick one of a select few options. This means that when one radio button is selected the others will become unselected. Thankfully, Maya does this for us. However, we need to tell Maya which radio buttons go together. In order to do this, we create what is called a 'radioCollection'. This is, essentially, a group of radio buttons that work together as a unit. To start the radioCollection, we call the radioCollection command. This opens-up our group to accept radioButtons. All radio buttons that are created before the next call to *radioCollection* will be added to the group. The only argument that we are going to use is the name of the radioCollection. This is very important so that we can later query to find out which of the radioButtons are selected. We can now call the *radioButton* command as many times as we need to, and this will get all of the options out there for the user. In this case, we need three. The only thing that is really new here are the -onCommand and the -offCommand. These flags execute whatever is between the double quotes, when a radio button is selected or unselected.

Next up, it is our old friend the checkBox:

```
rowLayout
        -numberOfColumns 2
        -columnAlign2 "left" "left"
        -columnWidth2 20 160;

text
        -label ""
        -width 10;

checkBox
        -label "Use Absolute Frame Number"
        -enable false
        absFrameNumberCheck;
//Back to column 2 columnLayout
setParent saveColumn2;
```

This works just like the previous checkbox. We want to left justify the checkbox and it's label. We setup a rowLayout for it and then add in the checkBox. The only difference being that we used the *-width* flag instead of the *-height* flag. This width flag is used to indent the UI element. We are going to use a text control again to take-up some of that space. So we use the *-width* flag to define the width of the spacer. If you look closely, there a new option for the checkBox that we have not seen yet. That is the *-enable flag*. This flag will set the state of the control to either be grayed out (disabled) or fully editable (enabled). To set this, we call the -enable flag and set it to either true or false. Try setting it to *true* and *false* and see the difference for yourself. Then we add the checkBox, and set the parent up to the *saveColumn2* layout again.

So far so good eh? Now may be a good time to take a break if you like, or you can power through to the end of this section. Currently, we have seen text controls, textScrollLists, buttons, checkBoxes, and radioButtons. There are only two more types of controls that we are going to use in this utility. Those are number fields and text fields. All of these elements are very well explained in the MEL command reference. Do not panic if you do not have them memorized!

```
//Layout to hold the indented "Start Frame" text and number
//fields
rowLayout
        -numberOfColumns 3
        -columnWidth3 22 60 20;

text
        -label "";

text
        -label "Start Frame";

intField
        -min 0
        -value 0
        -enable false
        saveStartFrameField;

//back to column 2 columnLayout
setParent..;
```

The first couple of lines should look pretty familiar to you. We setup a rowLayout, add a spacer and a label. After the label, there is a new control, the *intField*. All this basically does is give us a field that will only accept integer numbers. There are several flags that I am setting here to properly setup the *intField*. The first of these is the *-min* flag. If you look in the help, you will notice that this will set the lowest number that the field will accept. If you try to enter in a number that is less that what is specified by the -min flag, then it will reset the number to the minimum that you have just set. The next one is the *-value* flag. This flag sets the default value for the field. This means that once the control is displayed, Maya will set the value that is shown to whatever you specify by the -value flag. In this case, we are setting it to 0. The last flag, is the *-enable* flag. This works exactly the same as the flag for the checkBox from earlier. Lastly, we have the name of the field. This is very important because this is the name that we will use to check the value stored in the field or to set the value in the field later on in the script. Finally, we set the parent up a level again to get out of the rowLayout used to place the *intField* in the correct location. Now we have a nearly identical setup to add in the "'End Frame' intField:

```
//Layout to hold the indented "End Frame" text and number fields
rowLayout
        -numberOfColumns 3
        -columnWidth3 22 60 20;

text
        -label "";

text
        -label "End Frame";

intField
        -min 0
        -v 0
        -enable false
        saveEndFrameField;

//back to column 2 columnLayout
setParent saveColumn2;
```

Everything here is pretty much exactly the same. We only have a few more controls to go over in the 'Save' tab! Yay!

Next up, in the design, are a space and a couple of radio buttons. Let's add those now:

```
rowLayout
        -numberOfColumns 1;

text
        -label "Store Values as:";

setParent..;

columnLayout
        -columnAlign "left"
        -adjustableColumn true;

  radioCollection valueTypeCol;
  radioButton -label "Relative Transforms(local space)" ...
   -enable true relKeyValue;
  radioButton -label "Joints Only" -enable true bakeKeyValue;

setParent..;
```

There is not much new here, as we make a label for the radio buttons, which I wanted to be aligned left, as opposed to aligned center, like the top columnLayout is set to. We need to create a rowLayout and display the text. Then set the parent up a level again. Now we need a columnLayout to hold the radio buttons. Once that is setup, we call the *radioCollection* command again. Recall from ealier, if we were to simply add more radio button, they would be part of the same collection as the 'File Type' radio buttons. Since we want them to be part of their own collection, we need to make a new collection by calling *radioCollection*. We are going to call this on 'valueTypeCol'. Let's add the new radio buttons, just like before, and then set the parent back up a level again. You may notice, the next bit of code is slightly different:

```
//Now set the default value - set the collection in edit mode

//This has to be set after the intFields and abs/rel radio
//buttons have been created.

//select the first option
radioCollection -edit -select savePoseOnly saveTypeRadioCol;

//select relative as default
radioCollection -edit -select relKeyValue valueTypeCol;
```

If you recall, from way back when we were first creating the *window* to hold our UI, we set the window command in *query* mode. There are thee modes that a command can be set to. This example uses the *edit* mode. This allows us to modify settings for some of the controls. While *-query* allowed us to get information from an item, *-edit* allows us to change things about that item. In this case, we want to pre-select some of the radio button options for the user. To do this, we must call the *radioButton* command and use the *-edit* flag to set it into edit mode. Then we use the *-select* flag, followed by the radio button name of the element we want to select and lastly the name of the collection where we can find the radio button. The above code selects the 'savePoseOnly' radio button from the 'saveTypeRadioCol' collection. It also selects the 'relKeyValue' radio button from the 'valueTypeCol' collection. We will get into many more uses for *-edit* with all of the controls a little later on. If you feel like being a bit adventurous, take a look back at some of the strings between the double quotes after the *-command* flags for some of the buttons. You will now, probably be able to decipher a bit more of what is going on there.

This brings us to the last two controls. One is a checkBox, which we have been over a few times already, and the final one is a textField:

```
//spacer
text
        -label "";
        -height 27;

//Column layout to hold the Quick Save check box

//If we didn't use a new layout this text would
//get centered to the column 2 column.

columnLayout;

//Quick Save check box
//When checked it will enable the text field, when unchecked it
//willdisable it.

checkBox
      -label "Quick Save To File"
      -onCommand "textField -edit -enable true quickSaveText;"
      -offCommand "textField -edit -enable false quickSaveText;\
                   textField -edit -text \"\" quickSaveText;"
      quickSaveCheck;


//Back to column 2 columnLayout
setParent saveColumn2;

//textField to hold the path to the quick save file - disabled by
//default

textField
        -text "";
        -editable true
        -enable false
        quickSaveText;

//Back to column 2 columnLayout
setParent saveColumn2;
```

Firstly, we have a spacer followed by a checkbox with a label. All of this has been seen before. However, after that, there is a new control called a *textField*. This is much like the *intField*, in that it gives the user a place to input information. The first flag is the *-text* flag, which will allow you to fill the textField with a string of your choosing, by default. I, however, do not want anything to be in there by default, so I simply set it to an empty pair of double quotes (""). Next up is the -editable flag, which simply sets the control to allow the user to edit what is in the field or not. Since this is the place where the user will enter a path name, we want this field to be editable. If we were to simply display the text to the user, we could set *-editable* to be false. Lastly, we have the -enable flag which works just like all of the other *-enable* flags for the other controls. It either sets the control to normal or grayed-out. Following that, we have the name of the field. This is very important as it is the only way we can tell Maya which field we want to either get data from (using the *-query* mode) or set data to (using the *-edit* mode). Finally, we set the parent back to the main ColumnLayout.

The interface control is now all the way back to the original tabLayout:

```
//Back to column 2 columnLayout
setParent saveColumn2;

//Back to "Save" main rowLayout
setParent..;

//Back to main tabLayout
setParent..;
```

We are done with the 'Save' tab! Congratulations on making it this far. I know there were a few times where I waved my hands and called it magic, but I promise that I will get back to explaining what is going on with all of those *-command* flags. For now, give yourself a pat on the back, play with some of the items in the UI, and take a break. When you get back, we will start working on the 'Load' tab.

The 'load' tab does not have anything new in terms of UI controls. We have already gone over every type of control that will be in this utility. You should feel pretty comfortable while looking at the remaining two tabs. If you still need help some of the flags, use the f1 key!

```
//****** Load Tab *******
columnLayout
        -adjustableColumn true
        load;

  //section1 of load tab
  rowLayout
          -numberOfColumns 2
          -columnWidth2 200 200
          -columnAttach 1 "both" 5
          -columnAttach 2 "both" 5
          loadSec1RowLayout;

    //Left List
    columnLayout
        -adjustableColumn true
        -columnAlign "center"
        loadLeftColSec1;

      text
        -label "Source Objects";

      textScrollList
        -numberOfRows 23
        -allowMultiSelection false
        -selectCommand   "jcUpdateTargetList(\
                         "loadSourceObjectsTextList\",
                         \"loadTargetObjectsTextList\",
                         $myTargetArray);"

loadSourceObjectsTextList;

  //Back to loadSec1RowLayout
  setParent..;
```

For this tab, we are going to start with a columnLayout. There are two reasons for this. The first reason is that a tabLayout can only have one direct child. We basically need to make a container that will hold the rest of the UI. The second reason is that we can put as many things into the column layout as we like because they stack on top of each other.

We also use the -adjustableColumn flag to make sure that the items in the columnLayout stretch to each side of the layout.

We need to make the two textScrollLists for the 'load' tab. The one on the left is going to hold the source objects that are read from a file that was saved using the 'save' tab. The list on the right will hold the custom mapping targets for each source object. The flags simply tell Maya that we want two columns and that there will be a five pixel buffer between the two. We give the layout a name so that we can later query the layout or edit it using the -query and -edit modes. This takes us to the first of the two textScrollLists, the left list. We must set up a columnLayout inside the rowLayout so that we can put a few elements in the same column. The first element is a text label for the list. Following that, we have the textScrollList itself. Since that is all that we want in that first column, we now go back to the parent. This is the rowLayout. Now we can start building the second column.

```
//Right List
columnLayout
        -adjustableColumn true
        -columnAlign "center"
        loadRightColSec1;

//Text label
text
        -label "Target Objects (source centric)";

textScrollList
        -numberOfRows 23
        -allowMultiSelection true
        loadTargetObjectsTextList;

//set parent back to "Load" main column layout
setParent load;
```

Nothing very special here. Again, we make a new columnLayout so that we can stack a couple of elements. The first is, of course, the text label that describes what the list is used for in the UI. The second item is the textScrollList itself. This is exactly like the left column. Lastly, we set the parent back to the 'Load' layout. This was the columnLayout that is just inside the overriding tabLayout. This will allow us to continue to put items lower in the UI.

```
//text under lists
rowLayout
        -numberOfColumns 1
        -columnAlign1 "center";
//quick directions on usage
text
        -align "center"
        -label "Select animation source item on left.  Add
                objects to list at the right \n
                 to copy the animation to.";
//Back to "load" - main layout
setParent..;
```

We are simply adding-in a bit of descriptive text to help the user to figure out what to do on this tab. Then we go back to the 'load' layout to continue to add to the UI.

```
//Spacer
text
        -label "";
        -height 20;

//one row layout with two columns - 4 buttons under lists
rowLayout
        -numberOfColumns 2
        -columnAttach 1 "both" 5
        -columnAttach 2 "both" 5
        -columnWidth2 200 200;

//Buttons on left
        columnLayout
                -adjustableColumn true
                -columnAlign "center";

                button
                        -label "Load Objects From File"
                        -command "...some code goes here..."
                        loadLoadObjectsFromFileButton;

                button
                        -label "Remove Object From List"
                        -command "...some code goes here..."
                        loadRemoveObjectFromListButton;

        //Back to rowLayout
        setParent..;
```

This is where we add the two buttons under each column. So what we will do is make a rowLayout with two columns. Each column has two buttons in it. We start with the rowLayout, and then make a columnLayout to serve as the left two buttons. Now we use the *button* command to create a button and give it a label. Again, there is the *-command* flag. Continue to ignore it for now. We will come back to all of the UI and make the buttons actually do something. We then give the first button a name of 'loadLoadObjectFromFileButton'. Then we simply add the second button underneath of it. This one is called 'loadRemoveObjectFromListButton'. The names of these buttons are pretty long, however they are pretty descriptive. Use your best judgment on keeping the names of the buttons a decent length. Since we will not be using those names very often, it make sense to keep them longer to help explain what the button itself does. The easier the code is to read, the less frustration you will have later when you are trying to debug it.

Now that the second button in the first column is made, we set the parent back to the rowLayout and start making the right hand column of buttons.

The right hand column is essentially the exact same thing. Make a columnLayout and then fill it with two buttons. One called 'loadAddObjectButton' and the other called 'load RemoveObjectButton'. Finally, set the parent back to the upper 'load' layout. Coming up next, we will take a look at a new UI command:

```
//Spacer
text
        -label "";
```

```
        -height 10;

//Dividing line
separator
        -style "in";

//spacer
text
        -label ""
        -height 10;
```

The first and last pieces are simply text controls being used as spacers. However, there is a new command in-between the two called *separator*. This command draws a type of dividing line across the layout that contains it. There are eight different styles of separators. Please have a quick peek in the help file and take a look at what they are. For this UI we are going to use the simple but effective 'in' style.

On to section two of the "load" tab, this section is used to set two of the more global options for how the animation is copied. This section contains two checkBoxes and a single radioButton collection. This is a rowLayout that is made up of two column layouts, much like the previous section.

```
//Section 2
rowLayout
        -numberOfColumns 2
        -columnWidth2 200 200
        -columnAttach 1 "both" 5
        -columnAttach 2 "both" 5
        loadSec2RowLayout;

//Column1

//Checkbox and radio buttons

columnLayout
        -columnAlign "left"
        -adjustableColumn true;

//rowLayout to properly place the checkbox
rowLayout;

checkBox
-label "Auto Map Objects"

-onCommand "radioButton -edit -enable true onErrorContinue;\
radioButton -edit -enable true onErrorCancel;\
radioButton -edit -enable true onErrorAsk;\
clear $myTargetArray;\
textScrollList -edit -removeAll loadTargetObjectsTextList;\
button -edit -enable false loadAddObjectButton;\
button -edit -enable false loadRemoveObjectButton;"
-offCommand "radioButton -edit -enable false onErrorContinue;\
radioButton -edit -enable false onErrorCancel;\
radioButton -edit -enable false onErrorAsk;\
button -edit -enable true loadAddObjectButton;\
 button -edit -enable true loadRemoveObjectButton;"

loadAutoLoadObjectsCheckBox;
```

```
//The off/on command flags are used to specify a command that is
//executed when the button is pressed.
//Notice that these commands take the form of a string. This
//string comprises several lines that edit other UI elements.
//The strings are colored purple.

//backto columnLayout
setParent..;
```

First, much like before, we start out with a rowLayout called 'loadSec2RowLayout'. Then we begin with the first column, which will be the first checkBox, and the radioButton collection below that. Since we are stacking things in a column, we will once again use a columnLayout. If we were to simply place the checkBox in the columnLayout, the box and label would be split apart to stretch the control across the entire column. Since this simply looks ugly, we will throw it in a very simple rowLayout with one column and put the checkBox in there.

We have seen these *-onCommand* and *-offCommand* flags before. Let's talk about these a little bit more. As mentioned before, the code between the double quotes for the *-onCommand* flag will get executed when the checkBox is checked by the user. The *-offCommand* code between the double quotes will get executed when the user unchecks the box. This is useful for many reasons, however the most useful reason for us, is that it gives us a chance to enable and disable other parts of the UI based on what options the user has selected. So let's see what these commands are really doing.

The *-onCommand* is calling seven different commands. The first one is a radioButton command. Here we are using the *-edit* flag to put the radioButton into edit mode. If you remember, this will allow us to actually change the values of the button as opposed to the *-query* command which only lets us retrieve values from the button. In this case, we are putting the button into edit mode and then setting the *-enable* flag to true. This will light up the radioButton and allow the user to interact with it. As you can see, this is being executed on the 'onErrorContinue' button that is further down in the UI. The next two commands are also radioButton commands, doing the exact same thing to two other buttons. Those two commands are enabling the last two radio buttons further down in the UI. The fourth command is the *clear* command. This basically takes an array and clears it out. I realize that we have not yet gone over arrays, but we will get to it in due time. For now, think of an array as a variable that can hold a list of values of the same type. The clear command simply removes all of the values so the array is holding nothing. The fifth command is the textScrollList command. This is setting the "loadTargetObjetsTextList" from above into edit mode. Then we issue the *-removeAll* command. This will take any items in the textScrollList and remove them. Much like if the user selected all of the items and then selected the 'loadRemoveObjectButton'. The last two commands are regular *button* commands. The buttons are being set into edit mode and then they are disabled by setting the *-enable* flag to false.

The *-offCommand* flag is nearly the exact opposite of the *-onCommand* flag. The commands here are almost exactly the same as the commands called in the *-onCommand* flag. The only difference is that for each of the buttons, the *-onCommand* flag was turning off, this one is turning them on and vice-versa. Finally, we give the checkBox a name of 'loadAutoLoadObjectsCheckBox', and set the parent back to the columnLayout so we can add the radioButtons and their text label.

Now let's take a look at the "On Error" layouts:

```
//layout for the "On Error" text (for it to indent correctly
//with a spacer)
rowLayout
        -numberOfColumns 2
        -columnWidth2 20 160;

text
        -label ""
        -width 20;

text
        -label "On Error:";

//back to columnLayout for this section
setParent..;

//indent radio buttons with spacer
rowLayout
        -numberOfColumns 2
        -columnWidth2 40 100;

text
        -label ""
        -width 40;

//columnLayout to properly place radio buttons
columnLayout;

radioCollection loadOnErrorCol;

radioButton -label "Continue" -enable false onErrorContinue;
radioButton -label "Cancel" -enable false onErrorCancel;
radioButton -label "Ask" -enable false onErrorAsk;

//select default setting
radioCollection -edit -select onErrorContinue loadOnErrorCol;

//Back to section 2 main layout
setParent loadSec2RowLayout;
```

We setup a rowLayout with two columns so that we can use the first column to indent the "On Error:" text label. To do that, we simply use a blank text label in the first column and set the width to the desired size. Then simply add the text label with the actual text we want the user to see into the second column. Once that is done, we go back to the original columnLayout again.

Now we make a new rowLayout that will hold the radioButtons. Again, we set it to use two columns so that we can indent the radioButtons a bit by using a blank text label in the first column. Once the proper indentation is set, we can start adding the radioButtons. To do that, we are going to need a columnLayout to stack the radioButtons in. Then we need to create a new radioCollection so that this particular group of buttons will work together as a group. We will call this new collection the "loadOnErrorCol" collection. Next, we add three simple radioButtons and set them to be disabled by default. Remember, when the checkBox from above is checked, it will enable these radioButtons. When it is unchecked, it will disable them again. The three buttons are called "onErrorContinue", "onErrorCancel", and "onErrorAsk". Next, we call the *radioCollection* command again, and select one of the radio buttons, so that we have a default setting for the user to start with. Finally, we set the parent back to the section two rowLayout 'loadSec2RowLayout', which completes the left column. All that is left is to add one more checkBox to the second column of section two.

```
//column2
        columnLayout
                -columnAlign "left"
                -adjustableColumn true;

//layout for Adapt for Scale Checkbox
                rowLayout;
                   checkBox
                      -label "Adapt For Scale Difference"
                      scaleAdaptCheck;

//Back to "load" main layout
setParent load;
```

For this, we again, use a columnLayout with a rowLayout inside of it so that the checkBox and the label for the checkBox do not 'stretch'. Inside the rowLayout we actually add the checkBox and call it 'scaleAdaptCheck'. Lastly, we set the parent back to the 'load' tab and complete section two. Onward to section three!

The third section is made up of a few text labels, a couple of radioButtons and an intField. Before we actually start section three, we are going to add some space and a divider just like we did for section two.

```
//make divider with space above and below it
columnLayout
        -columnAttach "both" 5
        -adjustableColumn true;

        text
                -label ""
                -height 10;

        separator
                -style "in";

        text
                -label ""
                -height 10;

//back to "load" main layout
setParent..;
```

This is exactly the same as last time. We use an empty text field as a spacer above and below the divider, and use the separator command with the -style flag set to "in" to actually add the dividing line. Now for the guts of section three:

```
//section3
        columnLayout
                -columnAlign "left"
                -adjustableColumn true
                loadSec3ColLayout;

        //show current source file type
```

```
//this will change based on what type of file is loaded
text
        -label "Current Loaded File Type: (no file loaded)"
        loadedFileTypeTextLabel;

//layout for ApplyTo text (indented with spacer)

rowLayout
        -numberOfColumns 2
        -columnWidth2 20 160;

text
        -label ""
        -width 20;

text
        -label "Apply To:"
        -width 160;

//back to section 3 columnLayout
setParent..;
```

Section three is the most simple of the first three sections. This one only has a single column that holds all of the controls. We start out with a columnLayout called 'loadSec3ColLayout'. Next, we add the text label to help aid the user in the usage of the tab. This text label is a bit different. If you will notice, we gave this text label a name, 'loadedFileTypeTextLabel'. The reason this one has a name is because that text itself will change depending on what type of file the user loads. If the text control has a name, then we can access the text itself later by putting the text control into -edit mode. Next is a rowLayout that is simply being used to indent the 'Apply To:' label, just like we did a few times before. Since we used a rowLayout to get the indentation to work, we now need to set the parent back to the columnLayout of section three so we can add the radio buttons under the text label.

```
//layout for indented radio buttons
rowLayout
-numberOfColumns 2
        -columnWidth2 40 100;

text
        -label ""
        -width 40;

columnLayout;

radioCollection loadApplyToCol;

radioButton      -label "Default Frame (frame saved in file)"
                 -enable false applyToCurrentFrame;

radioButton
-label "Target Frame Offset"
-enable false
-onCommand "intField -edit -enable true loadOffsetFrameField;"
-offCommand "intField -edit -enable false loadOffsetFrameField;"
applyToOffset;

radioCollection -edit -select applyToCurrentFrame
loadApplyToCol;

//back to rowLayout
setParent..;

//back to section 3 columnLayout
setParent..;
```

Again, we are going to use a rowLayout to indent the labels and the radio buttons. In the first of the two column of the rowLayout, we simply stick an empty text control to add horizontal space. Then we use a simple columnLayout in the second column to add the radioButtons. We will need to use yet another rowLayout to keep the radioButtons and the button labels next to each other. Then we start another radioCollection called 'loadApplyToCol'. This will be the collection for the next two radioButtons. Once the collection is established, we add out two buttons. There is nothing really new here. The first one is simply disabled by default, and the second one, which is also disabled by default, has and *on* and *off* flag that simply enable and disable the intField that is lower in the UI. Finally, we call the radioCollection command and preselect a default option for the user. If we do not do this, then none of the radio buttons will be selected when the UI is first created. Lastly, we go up two levels in the UI hierarchy and create the final control for this section, the intField.

```
//indented "New Start Frame" text and number field
rowLayout
        -numberOfColumns 3
        -columnWidth3 80 90 200;

text
        -label ""
        -width 60;

text
        -label "New Start Frame: ";

intField
        -v 0
        -enable false
        loadOffsetFrameField;

//Back to "load" main layout
setParent load;
```

We are going to use a rowLayout with three columns. The first column is simply going to be used to indent the label and the intField. The second column has the actual text label for the intField, and the third column has the intField itself. Again, we are using the *-v* or *-value* flag to set the default value in the field. Then we are using the *-enable* flag to disable the field by default, and finally give it a name of 'loadOffsetFrameField'. Lastly, as always, we set the parent back to the 'load' layout to finish our tab.

```
//Divider with space above and below
text
        -label ""
        -height 10;

separator
        -style "in";

text
        -label ""
        -height 10;
```

Just as before, we use this little bit of code to give our third and fourth sections a bit of room between each other. Now on to section four!

```
//section4
rowLayout
        -numberOfColumns 3
        -columnWidth3 150 200 200;

                text
                        -label ""
                        -width 150;

                button
                        -label "Copy Source to Targets"
                        -command "...some code goes here..."
                        loadCopySourceToTargetsButton;

                text
                        -label ""
                        -width 150;

//Back to "load" main layout
setParent load;

//Back to main tab
setParent..;
```

This one is very simple. First, we make a rowLayout with three columns, just like before. The first is again used to simply indent the button. The second column actually holds the button, and the blank text field holds the space in the third column. To finish it off, we set the parent back to the 'load' layout and then back to the 'load' tab itself. Not much going on with this one, and we have just completed the 'load' tab! Next up the "Batch" tab.

There is nothing new in the batch tab. It is very nearly a mirror image of the 'save' tab structure wise. It includes two columns with a few buttons, textFields, and a textScrollList in them. This tab is very straight forward. By now, building this tab should be a breeze for you.

```
//****** Batch Tab *******
//main rowLayout for the Batch tab
rowLayout
        -numberOfColumns 2
        -columnWidth2 200 200
        -columnAttach 1 "both" 5
        -columnAttach 2 "both" 5
        Batch;

//Column 1
columnLayout
        -adjustableColumn true
        -columnAlign "center"
        batchColumn1;

//spacer
        text
                -label ""
                -height 5;

//Source File text
        rowLayout
                -numberOfColumns 1
                -columnAlign1 "left";

                text
                        -label "Source File";

//Back to column 1 of batch tab
setParent batchColumn1;
```

Since this tab is split-up into two columns, we will start with a rowLayout with two columns. In the first column, we will use a columnLayout and set it to center everything. Next, a spacer is used to move the UI down a bit from the top so it does not feel quite so cramped. Then we can start in with the first text label. We are going to use a rowLayout to hold the text label so we can justify the label to the left side of the window. Then we set the parent back to the batchColumn1 and continue down the UI.

```
//centered textField and Button
        rowLayout
                -numberOfColumns 2
                -columnWidth2 20 180;

                text
                        -label "";

                textField
                        -text ""
                        -width 150
                        batchSourceFileText;

//back to column 1
setParent batchColumn1;

        rowLayout
                -numberOfColumns 2
                -columnAlign2 "center" "center"
                -columnWidth2 20 180;

                text
                        -label "";

                button
                        -label "Select Source File"
                        -command "...some code goes here...";

                        -width 150
                        batchSelectSourceFileButton;

setParent batchColumn1;
```

This creates the first text field and button pairs. We are going to use a rowLayout for both the textField and the button so that we can limit

their width. We do not want these to be the entire width of the column because we want the importance stressed on the most used buttons where are at the very bottom of the UI.

```
columnLayout
        -adjustableColumn true
        -columnAlign "left";

//spacer
text
        -label ""
        -height 20;

setParent batchColumn1;

        rowLayout
                -numberOfColumns 1
                -columnAlign1 "left";

                text
                        -label "Log File";

setParent batchColumn1;

        rowLayout
                -numberOfColumns 2
                -columnWidth2 20 180;

                text
                        -label "";

                textField
                        -text ""
                        -width 150
                        batchLogFileText;

//back to column 1
setParent batchColumn1;

        rowLayout
                -numberOfColumns 2
                -columnAlign2 "center" "center"
                -columnWidth2 20 180;

                text
                        -label "";

                button
                        -label "Select Log File"
                        -command "";
                        -width 150
                        batchSelectLogFileButton;

//back to column 1
setParent batchColumn1;
```

The second pair works exactly the same way. We start with the label, then make the text field, and finally create the button. We also use a rowLayout to limit the overall width of the textField and button in exactly the same was as the first textField button pair. Next up are the radio buttons that let the user set the options for the batch process logging.

```
text
        -label ""
        -height 20;

rowLayout
        -numberOfColumns 2
        -columnWidth2 20 100;
text
        -label "List Type:";
        setParent..;

rowLayout
        -numberOfColumns 2
        -columnWidth2 20 100;

text
        -label "";

columnLayout;
//Batch List collection
radioCollection batchListTypeCol;
        radioButton
                -label "Files"
                -enable true
                -offCommand "clear $myBatchArray;\
                textScrollList -edit -removeAll batchFileList;"
                batchListFiles;
        radioButton
                -label "Folders"
                -enable true
                -onCommand "checkBox -edit -enable true ...
                batchRecursiveCheckBox;"
                -offCommand "checkBox -edit -enable false
                                batchRecursiveCheckBox;\
                                checkBox -edit -value false
                                batchRecursiveCheckBox;\
                                clear $myBatchArray;\
                                textScrollList -edit
                                -removeAll batchFileList;"
                                batchListFolders;

        //select Default setting
        radioCollection -edit -select batchListFiles
                        batchListTypeCol;
        setParent..;
        setParent..;

rowLayout
        -numberOfColumns 2
        -columnWidth2 40 100;

text
        -label "";

        //use recursion?
        checkBox
                -label "Recursive"
                -enable false
                batchRecursiveCheckBox;

setParent batchColumn1;

//Log file options and indented radioButtons
text
        -label ""
        -height 20;

        rowLayout
                -numberOfColumns 1
                -columnAlign1 "left";

                text
                        -label "Logging Style:";

//back to column 1
setParent batchColumn1;

        rowLayout
                -numberOfColumns 2
                -columnWidth2 20 150;

        text
                -label "";

//proper layout for radio buttons so the text doesn't get
//centered to column 1
        columnLayout;

                radioCollection batchLoggingStyleRadioCol;
                        radioButton -label "Errors Only"
                                batchErrorsOnly;
                        radioButton -label "Verbose"
                                batchVerbose;
                //select the first option for default
                        radioCollection -edit -select
                                batchErrorsOnly
                                batchLoggingStyleRadioCol;

//back to column 1
setParent batchColumn1;

text
        -label ""
        -height 10;

        rowLayout
                -numberOfColumns 1
                -columnAlign1 "left";

                text
                        -label "Log File Save Pref:";

//back to column 1
setParent batchColumn1;

        rowLayout
                -numberOfColumns 2
                -columnWidth2 20 150;

        text
                -label "";


//proper layout so text doesn't get centered to column 1
columnLayout;

radioCollection batchLoggingPrefRadioCol;
radioButton -label "Append" batchAppendLog;
radioButton -label "Overwrite" batchOverWriteLog;

//select the first option for default
radioCollection -edit -select batchAppendLog
                batchLoggingPrefRadioCol;


//back to column 1
setParent batchColumn1;
```

This part is very straight forward. Nothing tricky going on here at all. We are simply stacking the different controls one right after the other. The checkBox is using a rowLayout so that we can indent it properly and that's about it for this section of it. Please also note the call to the radioCollection command to edit which radio button is selected by default. Next, we have the final two buttons in the column.

```
//spacer
text
        -label ""
        -height 26;

//Last two buttons for column 1
button
        -label "Add Item To List"
        -command "...Some code goes here..."
        batchAddFileToListButton;

text
        -label ""
        -height 5;
button
        -label "Remove Item From List"
        -command "...some code goes here..."
        batchRemoveFileFromListButton;

setParent..;
```

Again, not much going on here. First we have a spacer followed by a button. Then another spacer and finally another button. Pretty simple, yet effective. Then we set the parent up to the main rowLayout and start the second column.

```
//Column2
//item list and button
        columnLayout
                -adjustableColumn true
                batchColumn2;

                text
                        -label "Batch List";
```

```
                textScrollList
                        -numberOfRows 30
                        -allowMultiSelection true
                        batchFileList;

                button
                        -label "Start Batch"
                        -command "…some code goes here…"
                        batchStartBatchButton;

//Back to main window
setParent Batch;
```

This is also a very simple setup. Firstly, we setup the column with a columnLayout so that we can stack the controls. Then we add a spacer, the textScrollList itself, and finally the 'Go' button. That is really all there is to it. We send the parent back to the 'batch' tabLayout and complete the final tab.

```
//We are finally done defining the UI.  Now lets show it!
showWindow $myWindow;

}//jcAnimCS
```

Now that the final tab is finished, we need to actually show the window. We call *showWindow* and pass it the string variable that we used to store the name of the window when we created it with the *window* command. Remember that? That was quite a while ago. Please have a look-back if it is not still fresh in your mind. Once the window is shown, we need to close the global proc called jcAnimCS with the closing brace and a comment that tells us which proc we are closing. Aren't you glad we did that first thing? Just something to keep in mind, when you setup pretty much any block of code, whether it be a loop, a proc, or a conditional statement of some kind, it is usually best to put in the beginning and ending braces so that you do not forget them. That kind of bug can be very difficult to find sometimes.

Basically that's it! You have made a semi-functional UI that should show up when *showWindow* is called. In addition, you have actually created a complete script. Once you source it and call the main global proc, which, if you remember, shares it's name with the name of the .mel file, the window will pop up. Nice work! You might as well take a break, grab a drink and enjoy the spoils of all of this typing! When you get back, we will start making some of these buttons actually do something. Take your time, I've got all day.

■ **Creating the functionality for the "Save" tab.**

Let's start adding functionality to the buttons. Back to the 'save' tab we go! In the save tab, look for the button called 'addObjectToSaveListButton'. Let's make this button actually do something.

```
//Add object button

button
-label "<< Add Object To List"
-c "if(`checkBox -query -value addHiCheck`){select -hi;}\
$myNewObject = `ls -sl`;\
jcAddToList($myNewObject, \"saveObjectTextList\", 0);"
addObjectToSaveListButton;
```

Most of this you already know. The *button* command is called to actually create the button. Then there are some flags that modify how the button is created or what it does. In this case, we have the *-label* and the *-c/-command* flags. The *-label* flag is old hat by now. However the -command flag has been skipped over, until now. The *-command* flag tells Maya what code to execute when the button is pressed. This code is surrounded by double quotes after the *-c/-command* flag is used. In this particular button, we have an if statement all crammed into one line. Normally this is not good practice as it can sometimes make the code hard to read. In this case, it is not too bad, and it will not force us to look into another proc to find the code that is executed.

Here is how this works. The if statement is checking the value of the 'addHiCheck' control. If you scroll down in the code a bit, you will find that control. This is the check box to ask the user whether or not they want to add the entire hierarchy or not. To get the value of the checkBox, we need to put the checkBox into *query* mode. Then we use the *-value* flag to get the value of the checkBox. This will return either a *true* or *false* depending on if the box is checked or not. If it is checked, we then do what is in-between the braces which is a *select -hi*. If you look in the help, you will see that this command will take the current selection and select all of the children as well. After that happens, we call the ls command using the *-sl* flag. 'ls' is a command from the Unix operating system (if you are a little familiar with Unix or Linux type operating systems you will recognize some of the commands in MEL) that is used to list the files in a directory. In Maya, ls will return the names of objects. The *-sl* flag instructs Maya to return the names of the selected objects. In the help file for this command, you will notice that it returns a *string[]*. This means that it returns the data in the form of an array of strings. Up until now, we have not said much about arrays. Essentially, an array is a single variable that can hold many values. These values must all be of the same type, which means that it can hold all strings, or integers, or floats, but not any two at the same time. If you look a bit further up in the jcAnimCS proc, you will notice that I have declared a variable named *$myNewObject*. This is a string. After the variable name, there are a pair of brackets. Those square brackets indicate that the $myNewObject variable is an array. The word 'string', in front of it all, indicates that it will be an array of strings. We will get more into how we exactly store the data and then how to actually retrieve it in a bit when we talk about the *jcAddToList* proc. For now, just know that the *$myNewObject* variable is holding the list of objects that the ls *-sl* command is returning.

Finally, we call the *jcAddToList* proc. This is a procedure that will add the list of names to a textScrollList. It just so happens that we now have a list of names and a textScrollList that needs filling. Before we get into the procedure itself, there is one thing that is a little odd, the '\' character. If you recall,the backslash is called an escape character. The last time we saw it, we were using it to make a newline character "\n". Now we are using it to do something quite different. This time, we have it in front of a double quote (' " '). We need to do this so Maya does not think we are trying to end the *-command* flag. If you remember, we started it with a double quote, and when we are done giving it code to execute, we need to end it with a double quote. This gets to be a bit of an issue when we are trying to pass strings to a procedure because to signify a string we need to put it in double quotes. Maya gives us a way to get around this by putting an escape character before the double quote. This will tell Maya that this is not the end of the *-command* flag, but the start of a string. Now let's take a look at the jsAddToList procedure. Firstly, we have the comment block for the procedure:

```
/*********************************************
jcAddToList()
Arguments:
        string $myObjectList[] - array of object names
                                 to add to the list
        string $myList - name of the textScrollList to add
item(s) to

        int $ignoreDup - check for duplicate?

Returns:
        none

Notes:
        none
*********************************************/
```

As usual, we have the 'arguments', 'returns', and 'notes' sections. We will have a comment block like this for every procedure in this utility. These will really help you when you look back on the code so that you do not have to read too much code to figure out what the procedure is supposed to do.

This procedure is going to take three arguments, so we list them in the arguments section. Firstly, we have the array of items that we want to add to the textScrollList. Second, we have a string which is the name of the textScrollList that we want to add the items to. Lastly, is a sort of convenience thing. This is the *$ignoreDup* argument. It is meant to be either a 1 or a 0. We want this function to not add duplicates to the list. However, later on we may want to have duplicate names in a list, whether it be for this utility or another utility later down the road. So we can use this argument to allow duplicates to be added or not. Now let's take a look at the declaration of the procedure:

```
global proc jcAddToList(string $myObjList[], string $myList, int
$ignoreDup)
{//jcAddToList
}//jcAddToList
```

This proc is very similar to the first one that we did, the *jcAnimCS* proc. We are making this a global proc so that any procedure loaded into Maya can see it. Next, we give it a name, and in this case, it's *jcAddToList*. Finally, we declare what arguments the procedure takes. These arguments can be named anything that you like. They do not have to match the variable name or whatever you are using when you call the procedure, however they can be the same if you want. I understand this can be a bit confusing, but once you see how it all works together, it should make sense. Essentially, when you are declaring an argument, it is just like making any other variable. These names will only be used within the procedure that you declare them, and cannot be seen anywhere else. This is called a 'local' scope, which means that the variable name is local to this procedure only. The antithesis of this is the 'global' scope, which we have talked about briefly. Anything that has the word 'global' in front of it will be able to be see by any script loaded by Maya.

Now for the arguments themselves. We have an array of strings called *$myObjList*. We know it is an array because of the square brackets that follow the name, and we know that the array is of type string because of the reserved word 'string' in front of the variable name. We have a variable called $myList that is of type string, and lastly we have *$ignoreDup,* which is an integer. Any time you wish to call this procedure, you must pass in an array of strings, a regular string, and an integer, just like we did with the *addObjectToSaveListButton*. Let's look at the meat of the proc.

```
{//jcAddToList

string $allListItems[];
int $isDupe = 0;

//Step through the list of objects selected and
//append them to the list

for($eachObject in $myObjList)
        {//step through selection list

                //First grab the contents of the list so we can
                //check for duplicates
                $allListItems = `textScrollList -query ...
                                        -allItems $myList`;

        for($eachItem in $allListItems)
        {//check for dupes
                if($eachObject == $eachItem)
                {//set dupe flag
                        $isDupe = 1;
                }//set dupe flag
        }//check for dupes

        //Edit the list and append new item
        //if not a duplicate, add to list

        if($isDupe != 1 || $ignoreDup == 1)
        {//add to list
                textScrollList
                        -edit
                        -append $eachObject
                        $myList;
        }//add to list

        //reset the $isDupe Flag
        $isDupe = 0;
        }//step through selection list
}//jcAddToList
```

The first action that this procedure takes is a *for* loop. Up until now we have not discussed the different ways to 'loop' though things. The two main ways are *for* loops and *while* loops. There are also two types of for loops. We are using one of them in this procedure. This type of *for* loop is called a 'for-in' loop. How it works is like this:

```
for($eachObject in $myObjList)
```

First, you call the for command and pass in what you want to loop through. The first variable is a single variable that will hold the current item in the array. The second variable is the array itself. So each time through the loop, it takes the next item in the array and stores it in the first variable. So in our case, we are going through each index of the array *$myObjList* (which was one of the arguments from above) and

storing it in the *$eachObject* variable. This will run the code between the braces for the "for" loop once for each item in the array. So if the *$myObjList* array had "sphere1", "cube1", and "cone4" in it, then the first time the loop runs, $eachObject would be set to "sphere1". The next time through the loop *$eachObject* will contain "cube1", and so on. The nice thing about this is that we can do something with each item in the array without having to manually address each item individually. Let's see what happens each time through the loop.

Since this procedure is used to add text items to textScrollLists, we need to get the contents of the textScrollList so that we can add to it. What we need is a place to store the current items in the list. So we declare a variable at the top of the procedure called *$allListItems* and make it an array of strings. We use the *textScrollList* command and put it into *-query* mode. Then we get the list of items in the list by calling the -allItems flag, and finally tell Maya which textScrollList to look at by giving it the $myList argument that was passed to this procedure. This will ask the textScrollList for all of the items in that list and store them into the $allListItems array. Now we need to check for duplicates. Basically, all we are going to do is take the current item that is to be added to the list, and compare it to each item already in the textScrollList. To achieve this, we are going to use another 'for-in' loop. This time, we loop through the list of items that are already in the textScrollList. These items have been stored into the $allListItems array. The variable that we are going to use to store the current item from the $allListItems array is the $eachItem variable. Inside this loop, we are going to do a single compare using an if statement. If *$eachObject* (which is the current object that is to be added to the textScrollList) is the same as (==) *$eachItem* (the current object that is already in the textScrollList) then execute the code between the braces, otherwise, skip it. If the statement is true, then we do have a duplicate and we want to set $isDupe to be 1. The $isDupe variable is a boolean type, meaning it is true or false.

The next if statement is used to check two things. If *$isDupe* is not equal to (!=) 1, or if *$ignoreDup* is equal to 1, do what is within the braces. The 'or', comparative operator that is signified by the double pipes (||) means if either the first statement is true or the second statement is true, then do what is within the braces. The only time that the code within the braces will be skipped, is if both statements on either side of the || operator are false. If one or both of the statements are true then we execute the textScrollList command and set it into -edit mode. Then we use the *-append* flag to add the item stored in $eachObject to the textScrollList stored in the $myList variable. That will add the new item to the textScrollList. Once that is finished, we reset the $isDupe variable to 0, and begin comparing the next item. That is basically it. This procedure takes a list of items, a textScrollList name, and a 1 or 0 to handle the duplicates. Then it adds those items to the list, checking for duplicates if the *$ignoreDup* variable is set to 1. Once you understand what is going on within this procedure, you will find that it is actually not very complex, but it is quite useful. Since there are four textScrollLists in this utility, we are going to use the procedure numerous times. That about does it for the *addObjectToSaveListButton*. Take a minute and look back over what actually happens when you press the " Add Object To List" button in the UI. Once you have a good grasp on what is really going on, then you can continue. We are going to build on these concepts as we construct the rest of this utility.

Next up is the "Remove Object From List" button. This is even more simple than the button that adds the objects.

```
//Remove Objects button
button
        -label "Remove Object From List >>"
        -c "jcRemoveFromList(\"saveObjectTextList\");"
        removeObjectFromSaveListButton;
//This button calls an outside procedure.
```

When this button is pressed, it will execute, as usual, the code in the double quotes following the *-c/-command* flag. This time, we are calling the *jcRemoveFromList* procedure and passing in the name of the textScrollList that we want to remove the selected items from. Let's go have a look at the *jcRemoveFromList* procedure. First-up, as always, is the comment block:

```
/*************************************************
jcRemoveFromList()
Arguments:
        string $myList - name of textScrollList to remove
item(s) from

Returns:
        none

Notes:
        none
*************************************************/
```

This procedure only takes one argument and it is a simple string that holds the name of the textScrollList that we want to remove the items from.

```
global proc jcRemoveFromList(string $myList)
```

Next in the declaration, you can see that this is a global procedure with one argument being the *$myList* variable. Nothing really special here, just plane and simple.

```
{ //jcRemoveFromList
        string $mySelection[];
        //Query the textScrollList to see what is selected
        $mySelection = `textScrollList -q -selectItem $myList`;

        //Remove each selected item
        for($each in $mySelection)
                { //remove each item
                //Edit the textScrollList and remove selected
                        textScrollList
                        -edit
                        -removeItem $each
                        $myList;
                } //remove each item

} //jcRemoveFromList
```

The contents of the procedure are pretty straight forward as well. The first action that we take is getting the names of the selected items in the list. To do this, we call the *textScrollList* command and set it into *-q/query* mode. Then we use the *-selectItem* flag to return the list of items that are selected. Lastly, the name of the textScrollList that we want to remove the items from is stored into the *$myList* variable. This is the only argument for the procedure. The results are stored in the *$mySelection* variable. If you look at the very top of this procedure, you will see that we have declared the *$mySelection* variable to be an array of strings.

Now that we have the names of the items we wish to remove from the list, we can loop through each of the names and selectively remove them from the textScrollList. To do this, we are again using a 'for-in' loop. This time, we loop through the array called $mySelection and each item of the array will be stored one at a time into the $each variable. Each time through the loop, the contents of $each will be updated with the next item in the array. To do the actual 'removing' of the item, we need to again call on the textScrollList command and this time set it in *-edit* mode since we are actually changing values. Once in -edit mode, we then use the *-removeItem* flag to remove the name that is stored in *$each*. Once this loop completes, all of the selected items in the list will be removed form the textScrollList.

The next button is by far the most complex in this tab. This is the "Go" button, or the "Save To File" button. This is where all of the true magic happens. Hold on to your hats, this is going to be a wild ride!

```
//Save to file Button on left

button

    -label "Save To File"
    -command "$filePath = `textField -query -text quickSaveText`;\
                                jcSaveAnimFile(\"saveObjectTextList\",\
                                                        $filePath,\
                `radioCollection -query -select saveTypeRadioCol`,\
                    `intField -query -value saveStartFrameField`,\
                      `intField -query -value saveEndFrameField`,\
                    `checkBox -query -value absFrameNumberCheck`,\
                  `radioCollection -query -select valueTypeCol`);"

                                saveToFileButton;

//This button command looks very complex, but please don't
//panic!
```

Honestly, the button part of this whole bit is not very complex. It simply grabs the text from the *quickSaveText* control and then calls the *jcSaveAnimFile* procedure with all of it's arguments filled in. Granted, there are seven arguments for this procedure, however it is not very complicated when you look at it in small chunks. Nothing really new so far, just more of it than normal. So let's dive right into the *jcSaveAnimFile* procedure. When you first look at this procedure, I am sure you are going to look at all 677 lines of it and maybe start to panic a bit. Try to remain calm, and remember that its all pretty simple once you break it down into small bits. So lets start off by looking at the comment block:

```
/*************************************************************
jcSaveAnimFile

Arguments:
string $myList - the Name of the textScrollList to get
                                the object names from
string $myPath - This contains a path to save the file to
If no path is specified (ie. "") then it
will display a prompt asking for the path
string $fileType - "savePoseOnly" or "saveAnimation"

int $rangeStart - the first frame of the animation

int $rangeEnd - the last frame of the animation

int $absFrame - 1 to store frames as their actual frame numbers
0 to store frames shifted so the first frame is 0

string $valueType - bakeKeyVal or relKeyVal - how to store the
values

Returns:
        none



Notes:

This proc saves out the pose info or the animation info to a file
specified by the user either in the save file dialog or through
the quickSave field.
*************************************************************/
```

As usual, we have the list of arguments that this procedure is looking for. It is looking for four strings, and three integers, just not in that order. The first is the name of the textScrollList that holds all of the names of the items that we want to save the animation information of. This is called *$myList*. Next, the *$myPath* argument holds the path to the file that we are going to save the animation information to. There is the *$fileType* argument that needs to contain either 'savePoseOnly' or 'saveAnimation'. These names are derived from the radioButtons on the 'Save' tab. If you look back to the button that is calling this procedure, you will see the *radioCollection* command called that is querying the radioCollection to see which button is selected. Next is the *$rangeStart* integer which holds the first frame of the animation that is to be saved. Following that, of course, is the *$rangeEnd* variable which specifies the last frame of the animation to be saved. The *$absFrame* is next, which needs to be set to either a 1 or a 0. This tells the procedure to either use the actual frame values from the animation or shift them all down so that the first frame starts are 0 instead. Lastly, we have *$valueType* which is set using the last set of radioButtons on the "Save" tab. This needs to be set to either "bakeKeyVal" or "relKeyVal". Basically, this tells the script to save the data from the channel box, or to get the rotation values of the joints only. That's it reall. Pretty simple so far eh? Here is the actual declaration of the procedure:

```
global proc jcSaveAnimFile(string $myList, string $myPath,
string $fileType, int $rangeStart, int $rangeEnd, int $absFrame,
string $valueType)
```

See, not so bad when you have the comments to look at first. Next in the proc, we have the variable declarations that are local to this procedure. We are going to skip these for now, and declare them as we need them later on in the procedure.

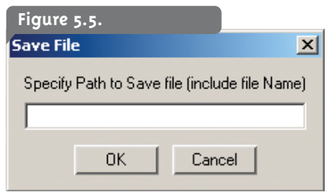
```
//If QuickSave is not on, create our own file save dialog as
//there is none built into Maya.
//This will return which button is pressed to dismiss
```

```
//the dialog

//Open a save file dialog

if($myPath == "")

{//Show dialog
        $savePromptDialog = `promptDialog -title "Save File"
        -message "Specify Path to Save file (include file Name)"
        -button "OK"
        -button "Cancel"
        -dismissString "Cancel"
        -defaultButton "OK"`;
```

This is the first action that is really taken. The quickSave text field was designed into the UI so that the user could simply type in a file path and not have to do it again as long as the checkBox was checked. This is nice when you are trying to edit the list of objects that you want to save. What this bit of code does, is check to see if there is a path typed into the quickSave field and take the proper action. If the field is empty (stated by the pair of double quotes with nothing between them), then we need to ask the user where they want the file to be saved. To do this, we are going to use the *promptDialog* command in Maya. If you look into the help file for this command, you will see that it will create a dialog window with a message to the user, a text field where the user can enter data, and a number of buttons to dismiss the dialog with. This is exactly what we needed. You will also notice that the return value of this command is a string that indicates how the dialog was dismissed, which we will get into in a bit.

The *promptDialog* command takes a few flags to be useful. We set the title of the dialog with the *-title* flag. Then we set the message that the user sees with the *-message* flag. Next, we add two buttons, one being "OK", and the other being "Cancel". The *-dismissString* flag returns what is in between the double quotes when the user used the "X" in the corner of the dialog window to close it as opposed to using "OK" or "Cancel". Lastly, we have the *-defaultButton* flag. This simply sets the focus onto a default button, which in this case is set to "OK". When executed it will look like this: FIGURE 5.5.

Figure 5.5.



Now we need to take what the user did, and get the path from the promptDialogBox.

```
        //Deal with response or QuickSave feature
        if ($savePromptDialog != "Cancel")
        {//get path
                //Get textField result from Prompt
                $myFilePath = `promptDialog -q`;
        }//get path
        else
        {//set path to nothing
                $myPath = "";
        }//set path to nothing
}//Show dialog
```

```
else

{//use $myPath

        $myFilePath = $myPath;

}//use $myPath
```

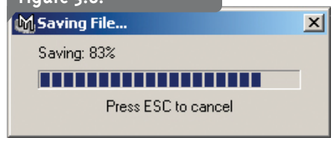
The first thing we need to do is check to see if the user simply canceled the dialog. If so, then we are done and do not need to actually do anything. However, if the *$savePromptDialog* variable does not contain the string "Cancel", then we need to get the new path. To do this, we are going to call the *promptDialog* command in *-query* mode. This simply returns the text from the text field into the promptDialog. We are going to store that into *$myFilePath*. This is something we have not talked about yet. This is the *else* statement. These follow *if* statements and provide an alternate execution path. So in this case if *$saveProptDialog* is equal to "Cancel" then the if statement is false and it goes to the else statement. So if the if statement fails, the execution will go to the else statement that follows and execute the code between the braces for the else statement. In this case, if the user cancels the dialog box, then we set *$myFilePath* to be blank. The first if statement works the same way. If *$myPath* does not equal "", then we drop down to the else statement that goes with that if statement and set *$myFilePathto* be the contents of *$myPath*.

Now that we have a path, let us continue down the procedure.

```
if($myFilePath != "")
{//save the file

        //Open File for Write

        //$myFileId holds the internal ID number for the new file

        $myFileId = `fopen $myFilePath "w"`;

        //Display the saved file type to the user
        print ("FileType: " + $fileType  + "\n");

        //open up progress window
        progressWindow
                -title "Saving File..."
                -progress $progressAmount
                -status "Saving: 0%"
                -isInterruptable true;
```

Here we do one final check to make sure that the path is not still blank. If it is, we skip the entire file saving process, otherwise we continue with the script. Next, we take this non-blank path and call the *fopen* command. The command will instruct Maya to open a file. In our case, we want to open the file for writing so that we can fill the file with the animation information, that is what the "w" bit is all about. It simply means that we want to open the file in "write" mode. Once that *fopen* command has completed, it returns the "file ID". This is an internal ID that Maya uses to keep track of each file. We are storing this ID into the *$myFileId* variable so that we can access the file later. The next little bit is all about giving the user feedback. First we use the *print* command to send some text out to the script editor letting the user know what type of file is being saved. Then we open up a "progress window". This is a little window that has a progress bar that we can update to show the user how far along in the process we are. To do this, we call the *progressWindow* command. We then use the -title flag to set the title of the window, much like with the promptDialog. The *-progress* flag sets the amount of progress already finished. This is what the window uses to set the width of the progress bar itself. The *-status* flag is simply a flag that we can use to give messages to the user. It is really nothing more than a label for the *progressWindow*. Lastly, the *-isInterruptable* flag tells Maya whether or not the progress of the current script can be canceled using the "Escape" key on the keyboard. Once all of the options are set, the progress window looks like this: FIGURE 5.6.

Figure 5.6.



Finally, we are at the point where we can actually start writing information out to the file! The first thing we need to do is check to see what kind of file we need to write:

```
if($fileType == "savePoseOnly")
        {//write out pose file
        }//write out pose file
```

We are going to tackle this procedure in two parts. The first part will be saving out the "Pose" only and the second part will be saving out the "Animation". So in the if statement above, we check to see if the user wants to write out a pose file. The *$fileType* variable is one of the arguments that are passed to this procedure. If you look back at the "Save To File" button, you will see where that information is actually coming from:

```
`radioCollection -query -select saveTypeRadioCol`
```

So for now, we are going to assume that the user has selected to save the pose only.

The first thing we need to do is write out some header information to the file so that when we read the file back in to apply the animation information to a character, we know exactly what kind of file we are dealing with.

```
//Write out header to the file with fprint "filePrint"
fprint $myFileId "POSE\n";
```

To actually write things out to a file, we will use *fPrint*. *fPrint* takes two arguments, the ID of the file to write to, and a string to actually write out to the file. So we give it *$myFileId* as the ID of the file we want to write to, and we want to write out "POSEn". This will put the word 'POSE' as the first line of the file and then add a newline character. At this point, if we close the file (which needs to happen in order for the data to be saved) and open it in a text editor you will see POSE as the first line and if you move the cursor to the end of the file, it will be sitting one line below the POSE line. Pretty simple really. The next thing we want to do is write out what type of values we have saved.

```
//Write out value type (absolute or relative)
if($valueType == "bakeKeyValue")
{//do absolute
```

```
fprint $myFileId "valueType: BAKED\n";

}//do absolute

else if ($valueType == "relKeyValue")
{//do relative
        //Print to file.
        fprint $myFileId "valueType: RELATIVE\n";
}//do relative
else
{//unknown type
        error ($valueType + " is an unknown value type!\n");
}//unknown type
```

Now that we are sitting at the second line of the file, we want to write out the value type into the "header". So we simply compare *$valueType* to "bakeKeyValue" and "relKeyValue". If it matches one of those, then we write the string "BAKEDn", or "RELATIVEn" to the second line of the file. If by chance it is neither of those types, then we send an error to the user stating that the current type is an unknown type. In this utility, you should never see this error, however if you are using this procedure in another utility then it is a very real possibility that someone may pass in the wrong information.

Next up we have the real meat of this procedure. We are going to use a "for-in" loop to loop though all of the objects in the textScrollList and save out their pose information.

```
//List Attributes for each object in PoseList
for($each in $myItems)
{//Do each object in list
}//Do each object in list
```

Just like before, the *$myItems* variable is an array of strings, and the *$each* variable contains the next name in the array each time through the loop. Now that we have an object name to work on (stored in *$each*), we can start getting information from it.

However, before we do any of that, we want to give the user a little feedback on what is going on. So let's update the progress window each time through the loop to show the user how much of the process has been finished:

```
//Determine Progress Amount
$percent = (((float)$progressCount)/(`size $myItems`));
$progressAmount = ($percent * 100);

//update progress window
if(`progressWindow -query -isCancelled`)
{//warning
     warning("Not all items were saved! \(user cancelled\)\n");
     break;
}//warning

progressWindow
        -edit
        -progress $progressAmount
        -status ("Saving: " + $progressAmount + "%");
```

This first line is a bit more complex than it appears at first glance. What we want to do, is take the total number of times that we are going through the loop and divide the number of times we have already been through the loop by that value. This will get us the percentage of completion so that we can update the progress window to show that percent. *$progressCount* is simply a integer that we are going to add 1 to each time through the loop. This will keep count of how many times we have been through the loop. Now, the (*float*) part before the $progressCount is something called a type cast. Basically what this is doing is taking the integer *$progressCount* and making it look like a float. So essentially all it does is take whatever whole number is in *$progressCount* and adding a .0 to it. In the Maya help file, you will find the entire table of type conversions. At some point, have a look at it so that you can understand what types can be converted to which other types. For instance, it would be nearly impossible to convert the string "Maya is great!" to an integer. However you could take the string "1235" and convert it to an integer with no troubles. The main reason that we are converting the *$progressCount* variable to a *float* is so that when the divide is finished, Maya does not make the result an integer by mistake. If this happens, the result will always be 0 because converting 0.15 to an integer simply truncates the number at the decimal point. This type conversion guarantees us that Maya will do the right thing.

To get the total number of times we are going to go through the loop, we need to get the size of the *$myItems* array. Please note that this is the same array as used in the "for-in" loop that we are currently working through. To get the size, we simply call the size command and pass in the array that we want the size of. An important thing to note here is that arrays start at 0. So the first element in an array is 0, making a ten element array go from 0 to 9. However when you get the size of the array, it does not tell you the last index, it tells you how many items in the array there are. So an array that goes from 0 to 9, will return a size of 10.

Once we have the percentage, we store it into the *$percent* variable. From there, we multiply the percent by 100 to get a whole number percent that is better looking to the user. Then we move on to updating the progress window. Remember when we set the *-isInterruptable* flag to true? This is where we need to deal with that. Firstly, we check to see if the progressWindow has been canceled. If it has, then we use the *warning* command to display a warning out to the script editor stating that not all of the items have been saved properly. Then we call the break command. This command essentially stops execution of the script. Once we have checked that the progress has not been cancelled, we can update the progress window. To do this, we call the *progressWindow* command again, set it to *-edit* mode and then change the -progress flag to reflect the new value we just calculated. For just a touch of polish, we update the *-status* flag to give a numeric representation of the percent complete. Next, we need to check to see if the *$valueType* is "bakeKeyValue", and if so, make sure that the current object is a joint. If it is not, then we simply move on to the next object.

```
//joints only?

if($valueType == "bakeKeyValue")
{//make sure object is a joint
        if(!(`objectType -isType "joint" $each`))
        {//no joint, skip
            warning ($each + " is not a Joint, skipping...\n");
            continue;
        }//no joint, skip
}//make sure object is a joint
```

Here we check to see if we are dealing with the "baked" keys. If so, we want to make sure that we are dealing with joints only. So, in an *if* statement, we call *objectType* with the flag *-isType* "joint". This essentially returns true or false after checking the object, in this case, *$each* against a type string, which in this case is "*joint*". If the object is of the specified type, it will return false. The if statement is saying, if the object is NOT of the type "joint", then execute the code within the braces. In this case, we warn the user that the object is not a joint and then move on with the *continue* command. *Continue* essentially says skip the rest of the code in this loop and start with the next iteration of the loop. This is similar to *break*, except break stops execution, where as *continue* skips to the next run through the loop.

Now we want to get all of the attributes that are being used with the current object.

```
//Grab a list of attributes from the current item
$myAttribList = `listAttr -keyable -scalar -visible $each`;
```

For this we need to use the *listAttr* command. This will return a list of the attributes associated with the objects that satisfy the given criteria. In this case, we only want attributes that are keyable, scalar, and visible. To get these, we use the flags, *-keyable*, *-scalar*, and *-visible*. These are then stored into the *$myAttribList* array of strings. This array will now contain all of the names of the attributes associated with $each.

Now it's time to write the name of the object to the file:

```
//write object name to the open file
fprint $myFileId $each;
```

This is achieved with another call to the *fprint* command. However, this time there is no "\n" character at the end. The goal here is for the POSE files, we want each line to contain all of the attributes and values for one object. We have the header which has the file type and the value type. On the third line, we start with the object name. This is followed by the first attribute and it's current value. Each subsequent attribute is added until all of the attributes have been written to the file for that object. Then we move down to the next line and do it all again for each object.

```
//save rotation order for the entire object
fprint $myFileId (" rotateOrder " + `getAttr (($each + ...
                                        ".rotateOrder"))`);
```

We are going to make the first attribute the rotation order. This is an object level attribute and every object should have this attribute, so we want to get that down first. To get the actual value of an attribute, we use the *getAttr* command. This simply returns the value of the specified attribute. In this case, we are getting the .rotateOrder attribute from the object contained in *$each*.

As a quick deviation, let's talk about how you can "build" strings to work with commands. Let's say we wanted to print the contents of *$each* as well as some other text. We could do the following:

```
print ("This is a neat object: " + $each + "\n");
```

What that will do is print "This is a neat object: pCone1". So by using parenthesis around the string that you want to print, as well as the addition operator, you can "concatenate" things to the string. In the line of code from the utility above, ($each + ".rotateOrder") will add the ".rotateOrder" part to the end of the object name. If *$each* contained "pCone1" then the final string would be "pCone1.rotateOrder" which is the full name for the attribute that we want to get the value from by using *getAttr*.

We are going to step through each attribute and print the names and values out to the file:

```
//write each attribute name and value to the file
for($thisAttrib in $myAttribList)
{//Do each attrib
  if($valueType == "bakeKeyValue")
  {//do baked

  //determine if this joint is a root joint
  //if so get translation as well as rotation

  if($thisAttrib == "translateX"||$thisAttrib == "translateY"...
                                  ||$thisAttrib == "translateZ")
    {//do translate
      if(jcIsRoot($each))
      {//write out translate

        //write attrib name
        fprint $myFileId (" " + $thisAttrib);

        $myValue = eval("getAttr " + $each + "." + $thisAttrib);

        //write value

        fprint $myFileId (" " + $myValue);

      }//write out translate

    }//do translate
    else
    {//do this attribute

        //write attrib name

        fprint $myFileId (" " + $thisAttrib);
        $myValue = eval("getAttr " + $each + "." + $thisAttrib);

        //write value

        fprint $myFileId (" " + $myValue);

    }//do this attribute

}//do baked
```

This bit of code looks fairly complex on the surface, but as usual, it is pretty tame when you break it down. First, we want to establish a "for-in" loop that will loop through each of the objects in the *$myAttribList* and store each object one at a time into the *$thisAttrib* variable. Easy enough, nothing really new here. Now we want to check to see if the *$valueType* is the "bakeKeyValue" type. If so, then we need to do some special things to get the right data for the joints only. For most joints we only need the rotation information. However if we are dealing with the root joint, then we will need the position information as well. Since we are looping through each of the attributes, we are going to first check to see if the current attribute is a translate attribute. If it is, then we check to see if the joint is a root joint by calling the *jcIsRoot* procedure that we will write a bit later. Basically *jcIsRoot* returns a 1 or a 0 depending on if the object that was passed to it is a root joint or not. If it is a root joint, then we want to record the translate attribute to the file. Again, to do this we use the *fprint* command just like before. We first pass it the file ID, and then the string to add. Also again, we are "building" strings with the parenthesis and the addition operator. We write out the attribute name, which will be one of the translate attributes. Then we get the value of that attribute and store it in the *$myValue* variable. In this case, we are going to use the eval command as well as the *getAttr* command. What eval does is let you "build" a string and then execute the code in that string. So when the string is built it will read: getAttr pCone1.translateX. This is assuming that the current object is pCone1 and the current attribute is translateX. As I just stated, once this string is built, then eval will execute it. Finally, we add a space and then print out the contents of *$myValue*. That is it for the root joint. If the *$thisAttribute* is not a "translate" attribute then we need to handle the data a little differently. Fortunately, this is much more straightforward. We simply get the data and write it out. We execute the code inside the else statement. First, as before, we write out the attribute name using *fprint*. Then, just as before, we use the getAttr command and build the full name of the attribute using string concatenations. We then use eval to execute the commands in that string. The returned value from the eval command is stored into *$myValue*. Finally, we take that value and print it out to the file.

Now if the user is not using the baked joint animation method to write out the file, we need to handle the relative data.

```
else
{//do relative
        //write attrib name
        fprint $myFileId (" " + $thisAttrib);


        //save value
        $myValue = eval("getAttr " + $each + "." + $thisAttrib);

        //write value
        fprint $myFileId (" " + $myValue);

        }//do relative
```

Writing-out the "relative" data is very easy. We write-out the name of the attribute, then we get the value of that attribute and store it in the *$myValue* variable. Finally, we print the new value to the file, and we are done!

```
}//Do Each attrib

//Write End of Object Tag
fprint $myFileId " <objEnd>\n";

//update progress count
$progressCount++;

}//Do each object in list
```

```
//Close (and save) the open file
fclose $myFileId;
}//write out pose file
```

Once all of the attributes for the current object have been written to the file, we write out a sort of "end of object" tag that we can search for when we load the files back in. We use *fprint* to write that at the end of the file and finally add a "\n" character to get the cursor in the file onto the next line. We update the progress count so that we can give the user some feedback on how far along the loop is. Once each object is done, we close the file so that the data is actually written to disk. To do this, use the *fclose* command and pass it the file ID of the file we wish to commit to disk. And that is it! We have done it! We can now save out all of the needed information to save a pose to disk! Congratulations. Now we're programming! Let's check out the save animation section:

```
else if ($fileType == "saveAnimation" || $fileType == ...
                                        "saveAnimationRange")

{//write out animation file
 //Determine Start/End frames
 if($fileType == "saveAnimation")
 {//do all keys

  if($valueType == "bakeKeyValue")
   {//use playback range
     $firstKey = `playbackOptions -query -minTime`;
     $lastKey = `playbackOptions -query -maxTime`;
   }//use playback range
```

If we are not saving the "pose" file type, then we drop down to this first *else-if*. It is here that we check to see if *$fileType* is either "saveAnimation" or "saveAnimationRange". If so, we continue into the braces.

In order to save the animation, we need to determine what frames we want to get the animation information from. To find this out, we need to check to see if we are saving an animation range or just saving the animation information in the file. We use another if statement to check this. If *$fileType* is "saveAnimation", then we want to get the animation information for all of the keys in the scene. Once this is determined, we need to find out if we are baking the keys for the joints or simply retrieving the relative animation information from the scene. So if *$valueType* is "bakeKeyValue" then we are baking the animation of the joints. Since the joints themselves may not have any keyframes on them, we cannot check to see what the first and last keyframe of the scene is. To get around this, we will simply use the beginning and ending of the playback range. We can get this information by using the *playbackOptions* in *-query* mode and getting the value from the *-minTime* and *-maxTime* flags. We will store those values into the *$firstKey* and *$lastKey* variables. If we are not baking the joint's animations then we can get the first and last key in the scene by doing the following:

```
else

{//find first and last keys for each object

//find the first and last key in the list of objects

        $firstKey = `findKeyframe -which "first" $myItems[0]`;
        $lastKey = `findKeyframe -which "last" $myItems[0]`;
```

Maya has a command called *findKeyframe*. We will use this, along with the *-which* flag, to get the first and last keyframes of the current first object in the list. This is all well and good if the first object has the first key set and the last key set. But there is no guarantee of this, so we need to search through each object and get the first and last keys in the entire animation:

```
  for($each in $myItems)
  {//search list

        $tempFirstKey = `findKeyframe -which "first" $each`;

        $tempLastKey = `findKeyframe -which "last" $each`;

        if($tempFirstKey < $firstKey)
        {//set new first key
                $firstKey = $tempFirstKey;
        }//set new first key
        if($tempLastKey > $lastKey)
        {//set new last key
                $lastKey = $tempLastKey;
        }//set new last key

  }//search list
 }//find first and last keys for each object
}//do all keys
```

We use another *for-in* loop to loop through each of the objects in the $myItems array. We get the first and last keyframes of that object. With these keys, we can compare them to the previous first and last keys. If the first key is earlier than the previous first key, then we make the new first key, the one from the current object. The same goes for the last key. If the last key of the current object is later than the last key, we have stored in *$lastKey,* then we make *$lastKey* equal to the last key of the current object. This sounds more complicated to explain in English than it actually is in code. Take a minute and really think about what is going on above, and remember we are trying to find the earliest and latest key in the entire scene.

```
else
{//use range
        $firstKey = $rangeStart;
        $lastKey = $rangeEnd;
        if($absFrame == 0)
        {//use relative frame numbers
                $minusOffset = $rangeStart;
        }//use relative frame numbers
        else
        {//use absolute frame numbers
                $minusOffSet = 0;
        }//use absolute frame numbers
}//use range
```

This code is executed if the file type is not "saveAnimation". This means that we are looking for a range defined by the user. This makes our job very easy. All we really need to do is grab *$rangeStart* and *$rangeEnd* from the arguments that are passed in and set *$firstKey* and *$lastKey* equal to those values. We also want to check to see if the "Absolute Frames" checkBox is checked. If it is, we store the "offset" for later use.

The offset is basically how many frames we must shift the animation before saving it out. To get this number, we simply take the first frame of the range and use that as the offset. We will store this value into *$minusOffset*.

Now that we have the first and last frame that the user cares about, we can start to print the needed information to the file.

```
//go through each frame in the range
for($currentFrame = ($firstKey); $currentFrame <= ($lastKey);...
                                                $currentFrame++)
{//loop through each frame

   //Determine progress amount
   $percent = (((float)$currentFrame)/$lastKey);
   $progressAmount = ($percent*100);

   //update progress window
   if(`progressWindow -query -isCancelled`)
   {//warning
     warning ("Not all items were saved! \(user cancelled\)\n");
     break;
   }//warning

progressWindow
        -edit
        -progress $progressAmount
        -status ("Saving: " + $progressAmount + "%");
```

We are going to setup our main loop so that we can go through each frame and take whatever information that we need from the scene. This happens to be a good use for the second type of *for* loop. This loop is simply called a for loop. This is setup a bit differently from the *for-in* loop. For this kind of loop, we give it a starting count, a condition that needs to be met, and a value to increment (or decrement) the count by each time through the loop. Here is an example that you will see most often:

```
for($i = 0; $i <= 10; $i++)
```

All that this does is store 0 into the variable $i. Then each time through the loop, it checks to see if $i is less than or equal to 10. It then executes the code between the braces for the loop. Once it has reached the end of the loop, $i is inremented. ($i++ is the same as saying $i = $i +1;). Now $i is 1. It checks to see if $i is less than or equal to 10. In this case it is. So it runs the loop again, then adds 1 to $i. It will continue to do this until $i is greater than 10. Once it is, it stops running through the loop and picks up execution of the code on the line right after the closing brace for the loop. The actual code looks like this:

```
for($currentFrame = ($firstKey); $currentFrame <= ($lastKey);...
                                                $currentFrame++)
```

This is what is going on here. We set *$currentFrame* to be whatever *$firstFrame* ended up being (this was set earlier in the code when we were figuring out what the first and last frames of the animation were). Then we check to see if *$currentFrame* is less than or equal to *$lastKey*. If it is, we run through the loop. Once the end of the loop is reached, we add one to *$currentFrame* and run the check again. If *$currentFrame* is less than or equal to *$lastKey*, we run the loop again. If not, we skip the loop and pick up execution with the lines of code just after the closing brace for the loop.

Now, assuming that *$currentFrame* is less than *$lastKey*, the program enters the loop. The first thing that we want to do here is give the user some feedback. We setup the *$percent* variable to update the progress window, just like we did before. Then we run a check to see if the progress window has been cancelled. If it has, we warn the user that not all of the information has been saved and we stop execution. If it has not been cancelled, then we edit the values in the progress window to reflect the current progress. The only difference for this one is that progress is recorded in how many frames have been completed out of the total number of frames, where as before it was how many objects were completed out of the total number of objects.

```
//Setup main counter for preFileArray
//(this just keeps our place in the array if we are baking the
//joint rotations)

$preFileCount = 0;

//Loop through all objects in the myItems array

for($each in $myItems)
{//do each object
    if($valueType == "bakeKeyValue")
    {//make sure object is a joint
      if(!(`objectType -isType "joint" $each`))
    {//no joint, skip
       warning ($each + " is not a Joint, skipping...\n");
       continue;
     }//no joint, skip
}//make sure object is a joint
```

Once the progress window has been properly setup and we are inside the main loop, we need to set a *$preFileCount* variable. For now, do not worry about what this does, we will come back to it later. We want to start a loop that will walk through each object for this frame. Inside this loop is where we do the saving. Once inside this loop we check to see if we are baking the joint animations. If *$valueType* is equal to "bakeKeyValue" then the user wants to bake the joint animations. Just like in the "POSE" section, we need to check to see if the current object really is a joint. If it is not, we skip the object and continue on to the next object. We check this just like we did before using the *objectType* command. Once we have established that we are working with a valid object we need to write that object's name out to a file:

```
//write out object name
$preFileArray[$preFileCount] = ("\nObjName: " + $each);
$preFileCount++;
```

This is where the *$preFileCount* variable comes into play. What we want to do is setup an array of strings that is an exact copy of the file we want to write out. The reason that we do this is because if you remember, the only time the data actually gets written to disk when you are writing to a file is when you close the file. If the file is opened and not properly closed, data corruption can occur. To minimize the chance of the computer crashing or something while the file is open, we gather all of the information that we need into an array, and when we are all done, we open the file for writing, dump the entire array to the file one line at a time and then close the file. This keeps the amount of time a file

is open to a minimum. We are going to use the *$preFileCount* variable to keep track of which index in the array we are on. In the end, this index is going to directly correlate to the line in the file. Index 0 (which is the first item or element in the array) will be the first line in the file. Index 1 will be the second and so on. As you can see, the first thing that we do is put the object name into the array. Once that code has been executed, the array may look like this:

```
ObjName: pCone1
```

At this point, it only has one item in it and it's the line above.

Just as a note, if you have not yet looked at an animation file that this utility writes out, please take a moment, and save a simple animation out to a file. Open the file in a text editor so that you can kind of keep track of where we are in the file itself. Your file may look something like this:

```
ANIMATION
valueType: RELATIVE
ObjName: pSphere1
rotateOrder: 0
AttribName: visibility
keyVal: 0 1 * * * * * * * * * * * * * * * * * * * *
inTanTypes: 0 spline * * * * * * * * * * * * * * * * * * * *
outTanTypes: 0 step * * * * * * * * * * * * * * * * * * * *
inTanWeights: 0 0 * * * * * * * * * * * * * * * * * * * *
outTanWeights: 0 0 * * * * * * * * * * * * * * * * * * * *
inTanAngles: 0 0 * * * * * * * * * * * * * * * * * * * *
outTanAngles: 0 0 * * * * * * * * * * * * * * * * * * * *
AttribName: translateX
keyVal: 0 0 * * * * * * * * 5 -1.637610661 * * * * * * * * 10
6.326626051
inTanTypes: 0 spline * * * * * * * * 5 spline * * * * * * * * 10
spline
outTanTypes: 0 spline * * * * * * * * 5 spline * * * * * * * *
10 spline
inTanWeights: 0 1 * * * * * * * * 5 1 * * * * * * * * 10 1
outTanWeights: 0 1 * * * * * * * * 5 1 * * * * * * * * 10 1
inTanAngles: 0 -18.13476758 * * * * * * * * 5 32.32000688 * * *
* * * * 10 57.87912957
outTanAngles: 0 -18.13476758 * * * * * * * * 5 32.32000688 * * *
* * * * 10 57.87912957
```

At the top of the file you can see the two line header that says this is an "ANIMATION" file and that the values are "RELATIVE". The next line is the one we just put into the array where it stores the name of the object. Next we store the rotationOrder just like we did with the "POSE" file. After that we store the attribute name and all of the keyframe information that goes with it.

As I just mentioned, the next thing we want to write-out is the rotation order:

```
//write out rotation order
$preFileArray[$preFileCount] = ("\nrotateOrder: " + `getAttr
                                ((each + ".rotateOrder"))`);
$preFileCount++;
```

This grabs the rotation order of the current object, and then stores it into the *$preFileArray* which is where we are storing an exact copy of the file to write out later. Once we store the value, we increment the *$preFileCount* again to move into the next index of the array which will be the next line in the file.

```
//grab list of attributes for current item
$myAttribList = eval("listAttr -keyable -scalar -visible " + ...
                                                    $each);
```

Just like last time, we now need to get a list of the attributes that are associated with the current object. We are going to store this list into the *$myAttribList array*.

```
//loop through each attribute
for($thisAttrib in $myAttribList)
{//loop through each attribute

        //build name.attribute string for ease of use later
        $currentFullName = ($each + "." + $thisAttrib);
```

Now we want to loop through each attribute for the current object on the current frame. This would make three loops deep. Anytime that you have a loop inside of a loop, they are called "nested loops".

All this part of the code does is build a string that is the full name of the object with the attribute. Then stores that string into *$currentFullName* so that we can easily access the data.

```
//set this up incase we ever want to do something on a whole
//range of keys at once.
//right now however it will only act on 1 frame at a time.  Its
//always good to leave
//options for yourself later.
$currentRange = ($currentFrame + ":" + $currentFrame);
```

This part of the script is not strictly necessary. However, sometimes it's nice to put in features that you may want to utilize later. This is simply storing a string into *$currentRange* that is the current frame, then a colon, then the current frame again. So if the current frame is 10, then it would look like this "10:10". Many of the animation functions in Maya will work on a whole range of frames, and that is the format that they require. The start frame, a colon, and then the end frame for the range.

It is now time to get the "relative" values if this is what the user has requested.

```
if($valueType == "relKeyValue")
{//do relative

  //Make sure object.attrib is indeed animated
  if(`connectionInfo -isDestination $currentFullName`)
  {//It is animated

    //Determine what is animating this attribute (the driver)
    $animationDriver = `connectionInfo -sourceFromDestination...
                                        $currentFullName`;

//Break it up on the "."
$numTokens = `tokenize $animationDriver "." $driverTemp`;
```

```
//Save the driver name only, back to the $animationDriver

$animationDriver = $driverTemp[0];

//make sure that the driver is an animation curve
$myNodeType = `nodeType $animationDriver`;
```

Of course the first thing that we need to do is check *$valueType* and see if it contains "relKeyValue". If it does, then we move on and record the animation information. First we need to check to see if the object and the intended attribute are actually animated. To do that, we use the *connectionInfo* command and look to see if the attribute is the destination of a connection by using the *-isDestination* flag. If it is, then it is animated. Now that we have established that the attribute is indeed animated, we need to make sure that an animation curve is driving the animation. We can figure this out by using *connectionInfo* again and this time use the *-sourceFromDestination* flag. This will grab the source of the animation and store it into *$animatoinDriver*. At this point, if you were to print out the contents of *$animationDriver,* then you may see something like this:

```
pCone1_translateX.output
```

This is the actual "plug" that is driving the animation of the attribute. The only part of the that string that we are really concerned with is the name of the driver and not the .output part. So we are going to use a command called *tokenize* that we will end up using over and over again throughout this utility. Basically, this will allow you to take a string and break it up into parts based on a pattern. *Tokenize* takes three arguments, the first being the string that we want to break up, the second being the pattern in which to divide the string with, and the third is an array of strings, where each index of the array holds one part of the original string that had been "sectioned off" by the pattern. *Tokenize* also has a return value, that is the number of "tokens" or parts that were sectioned off by the pattern. So for instance:

```
//Break it up on the "."

$numTokens = `tokenize $animationDriver "." $driverTemp`;
```

This will take the string in *$animatoinDriver*, which we will pretend is "pCone1_translateX.output" and divide the string up by the pattern. The pattern, in this case, is a period (.) character. The results will be stored in $driverTemp. The first element in the array (or *$driverTemp[0]*) will have "pCone1_translateX" in it. The second element (or *$driverTemp[1]*) will have "output" in it. The pattern that was used to divide the string is discarded. Once that is finished, the integer "2" will be stored in *$numTokens* because that is how many resulting "tokens" there were.

We want to take the *$driverTemp* array and store the first element (*$driverTemp[0]*) into a regular string for ease of use. So we set $animationDriver equal to *$driverTemp[0]*. This is the first element in the array.

As a quick note on arrays, which we will get into more later, this, *$driverTemp[0]* is pronounced "driver temp sub zero". This is how you access data in an array. You can store it this way or retrieve it. So for instance we could say:

```
$driverTemp[5] = "Arrays are neat!";
```

Now the sixth element in the array contains "Arrays are neat!". Now we could print that back out by doing the following:

```
print ("My Value: " + $driverTemp[5] + "\n");
```

This would print out "My Value: Arrays are neat!" and then put the cursor on the next line. OK back to the utility...

Now that we have the driver name stored into *$animationDriver,* we need to check to see if the driver is actually an animation curve. For this, we are going to use the *nodeType* command. This command will return the "name" of the class that the node fits in. This class name will be stored into the *$myNodeType* variable for later use.

Next, we do the actual check to see if the driver is indeed an animation curve. This will start the next main block of code.

```
if(`isAnimCurve $animationDriver`)
{//good driver
```

The *isAnimCurve* command is really a script that ships with Maya. This script basically takes the node type and checks a part of the node type name and determines if it says "animCurve". If it does, then it returns true. Otherwise it returns false. So now that we have finally determined if what is driving the animation is in fact an animation curve, we can start to gather the keyframe information from the curve.

```
//check if there is a keyframe at this time
if(`keyframe -time $currentRange  -query -keyframeCount ...
                                              $animationDriver`)
{//record info
```

At this point, we should be all set. The only thing left to do is make sure that there is a key for this attribute on the current frame. If there is, then we will grab all of the needed information from it. To determine if a key is indeed on the current frame, we need to use the *keyframe* command. The first flag that we pass to it is the *-time* flag. Since we had set up the *$currentRange* variable earlier on, we are going to continue to use it here. This command can work on a single frame or a group of frames. So to keep things consistent, we are going to use the range instead of a single frame. Because we are setting the range to be a single frame, it would make no difference if we used the range of one frame or just the single frame number. Next, we set the command into *-query* mode and use the *-keyframeCount* flag to return the number of keyframes on the current frame. If the value returned is 0, then the if statement will fail and we will move on to the next frame. If it is not 0, then we record the information. Before we get into that, let us talk about what makes up a keyframe exactly.

A keyframe consists of a few things. To start with, there is the time of the key frame. This is where the key is on the timeline. Second, there is the actual value of the key. This is the position of the key in the vertical direction in the graph editor, which is simply the value in the channel box of the attribute that is being keyed. Lastly, there are the tangents for the key. Each key has six tangent parts. These are the "in" and "out" tangent types, tangent weights, and tangent angles. These directly correlate with the tangent handles in the graph editor. In order to properly preserve the animation, we need to make sure that we copy all of this tangent information along with the time of the key as well as the value of the key. The file format works as follows. First we record the

name of the object. Then on the next line, the rotation order for the object. This is followed by the attribute name. We store the value and all of the tangent information on a separate line. On each line, we start with the frame number, then a space, and then the value for each part of the key. If there is no key on the current frame then we put an "*" instead of a time/value pair. Let us start with the time and value:

```
//record key time
$keyVals = ($keyVals + " " + ($currentFrame-$minusOffset));

//save value
$valTemp = `keyframe -time $currentRange -query -a -vc ...
                                        $currentFullName`;
$keyVals = ($keyVals + " " + $valTemp[0]);
```

As I just stated, we need to get the frame number. To do that, we take the *$currentFrame* and subtract the *$minusOffset* from it. If you recall, the *$minusOffset* is brought into play if the "Use Absolute Frame Number" box is not checked. This will essentially renumber all of the frames starting with 0. Once we have the value, we take whatever is in *$keyVals* at the moment, add a space, and then add the frame number. We need to get the value of the keyframe, so once again we use the *keyframe* command. As before, we tell it what frame to look at using the *-time* flag. Then we set it into *-query* mode. In *-query* mode, we ask for the *-absolute* flag and the *-valueChange* flag. These flags will return the position of the key in the y (vertical) axis of the graph editor. This is the value that we are looking for. The *keyframe* command with those flags set will return an array of floats. We store those values into *$valTemp*. Lastly, we take what is already in *$keyVals* and add a space followed by the contents of *$valTemp[0]* which should be the value of our key. After we run though all of the frames, *$keyVals* should contain something like this:

```
//An example from the animation file.
keyVal: 0 0 * * * * 4 -1.637610661 * * * * * * 8 6.326626051
```

First we have the "keyVal" tag, then we have the first frame number which in this case is 0. The next number is the value of the key. Then we have a bunch of "*" pairs. This means that there are no keys on those frames. You will see we have a key on frame 4, and it has a value of -1.637610661. This is followed by more "*" pairs until we get to frame 8 which has a value of 6.326626051.

Now that we have the key values down, we need to deal with the tangent types, weights, and angles:

```
//inTanTypes
$inTanTypes = ($inTanTypes + " " + ($currentFrame ...
                                        - $minusOffset));
$inTanTypesTemp = `keyTangent -time $currentRange -query
                                -inTangentType $currentFullName`;

$inTanTypes = ($inTanTypes + " " + $inTanTypesTemp[0]);

//outTanTypes
$outTanTypes = ($outTanTypes + " " + ($currentFrame - ...
                                        $minusOffset));
$outTanTypesTemp = `keyTangent -time $currentRange -query ...
                               -outTangentType $currentFullName`;

$outTanTypes = ($outTanTypes + " " + $outTanTypesTemp[0]);
```

```
//inTanWeights
$inTanWeights = ($inTanWeights + " " + ($currentFrame ...
                                        - $minusOffset));
$inTanWeightsTemp = `keyTangent -time $currentRange -query
                                    -inWeight $currentFullName`;

$inTanWeights = ($inTanWeights + " " + $inTanWeightsTemp[0]);

//outTanWeights
$outTanWeights = ($outTanWeights + " " + ($currentFrame
                                        - $minusOffset));
$outTanWeightsTemp = `keyTangent -time $currentRange -query
                                    -outWeight $currentFullName`;


$outTanWeights = ($outTanWeights + " " + $outTanWeightsTemp[0]);

//inTanAngles

$inTanAngles = ($inTanAngles + " " + ($currentFrame
                                        - $minusOffset));
$inTanAnglesTemp = `keyTangent -time $currentRange -query
                                    -inAngle $currentFullName`;


$inTanAngles = ($inTanAngles + " " + $inTanAnglesTemp[0]);

//outTanAngles
$outTanAngles = ($outTanAngles + " " + ($currentFrame
                                        - $minusOffset));
$outTanAnglesTemp = `keyTangent -time $currentRange -query
                                    -outAngle $currentFullName`;


$outTanAngles = ($outTanAngles + " " + $outTanAnglesTemp[0]);

}//record info
```

The procedure for getting the tangent information is nearly identical to getting the value information. Firstly, we get the current frame number and subtract the offset. We then store that number to the end of the string that is holding the current information. Use the *keyTangent* command instead of the *keyframe* command to get the tangent information. Set the *-time* flag just as with the *keyframe* command. Then set it into *-query* mode and use the proper flag for the information that we want. For instance, when we want the "in tangent type", we use the *-inTangentType* flag. This is the same with all of the lines of code that retrieve tangent information. Once we have the value, we tack it on to the end of the string that is holding the tangent information. Once this is complete for all of the frames, you should have something that looks like this:

```
inTanTypes: 0 spline * * * * * * * * 5 spline * * * * * * * * 10
spline
outTanTypes: 0 spline * * * * * * * * 5 spline * * * * * * * *
10 spline
inTanWeights: 0 1 * * * * * * * * * 5 1 * * * * * * * * * 10 1
outTanWeights: 0 1 * * * * * * * * * 5 1 * * * * * * * * * 10 1
inTanAngles: 0 -18.13476758 * * * * * * * * * 5 32.32000688 * * *
* * * * * 10 57.87912957
outTanAngles: 0 -18.13476758 * * * * * * * * * 5 32.32000688 * * *
* * * * * 10 57.87912957
```

Now, remember how I said that if there was no key on the current frame for the current object and it's attribute, that we will place an "*" pair instead? This is probably one of the trickiest part of this script. Knowing how to script this type of information can be useful in many different scripts. This is how its done:

```
else

   {//no key

        //if there is no key on this frame, put a "*" instead

        //record key time

        $keyVals = ($keyVals + " *");
        $inTanTypes = ($inTanTypes + " *");
        $outTanTypes = ($outTanTypes + " *");
        $inTanWeights = ($inTanWeights + " *");
        $outTanWeights = ($outTanWeights + " *");
        $inTanAngles = ($inTanAngles + " *");
        $outTanAngles = ($outTanAngles + " *");

        //record key value

        $keyVals = ($keyVals + " *");
        $inTanTypes = ($inTanTypes + " *");
        $outTanTypes = ($outTanTypes + " *");
        $inTanWeights = ($inTanWeights + " *");
        $outTanWeights = ($outTanWeights + " *");
        $inTanAngles = ($inTanAngles + " *");
        $outTanAngles = ($outTanAngles + " *");

   }//no key

}//good driver
```

In the case where the *if* statement that checks to see if there is a key on the current frame fails, we drop down into the *else* statement and execute the above code. Just to refresh your memory, this is the *if* statement that checks to see if there is a key on the current frame or not:

```
if(`keyframe -time $currentRange  -query -keyframeCount ...
                                        $animationDriver`)
```

If there is no key, then we simply take the current string and tack on a space and a "*" for the times and a space and an "*" for the values as well. Later on, when we import the file, we will use those "*" pairs to skip to the next frame.

Lastly, there is a bit of error checking

```
else
{//driver not an animation curve.. skip
warning ("Driver: " + $animationDriver + " is not an ...
                                    animCurveTL node..  skipping\n");
continue;
}//driver not an animation curve.. skip
```

```
}//it is animated
else
  {//no animation.. skip
  warning ($currentFullName + " does not seem to be animated ...
                                                skipping...\n");
  }//no animation.. skip

}//do relative
```

If the driver for the animation is not an animation curve, then we warn the user. If the object does not seem to be animated, we warn the user and move on to the next object.

That is it! That's all there is to storing the "relative" key data. Next we are going to tackle the "Baked" joint rotations.

```
//Do the bake style
else if($valueType == "bakeKeyValue")
        {//do baked
                //update time
                currentTime $currentFrame;
```

If the *$valueType* is not "relKeyValue", then we drop down to the *else-if* statement and check to see if it is equal to "bakeKeyValue". If it is, then we start to save the joint rotations out to the file. The main difference between this system and the "Relative" key system is that the joints are not actually keyed. Because of this, we cannot use *keyFrame* to get the value. What we need to do is step through every frame of the animation and save the rotation values for each joint. The only exception to that is the root joint. For this, we need to also store the translation values. The concept of "baking" is just that. If you look at the `Edit->Keys->Bake Simulation` tool in Maya, you will see that it is very similar to what we are doing. We simply move the time slider to the next frame and record all of the values for each joint. Then we move to the next frame and do it all again until we run out of frames.

Start by moving the time slider to the *current time*. To do this, we use the currentTime command and pass to it the frame number that we want to set Maya to. Once this is set, we can begin to record the values:

```
//determine if this joint is a root joint
//if so get translation as well as rotation

if($thisAttrib == "translateX" ||
   $thisAttrib == "translateY" ||
   $thisAttrib == "translateZ")
{//do translate
if(jcIsRoot($each))
{//write out translate
//write attrib name
$keyVals = ($keyVals + " " + ($currentFrame-$minusOffset));
$tempVal = getAbsLoc($each, $thisAttrib);
$keyVals = ($keyVals + " " + $tempVal);

}//write out translate
else
{//skip this attrib

continue;
```

```
  }//skip this attrib

}//do translate
else
 {//do other attributes
    $keyVals = ($keyVals + " " + ($currentFrame-$minusOffset));
    $tempVal = eval("getAttr " + $each + "." + $thisAttrib);
    $keyVals = ($keyVals + " " + $tempVal);
 }//do other attributes
```

The first thing we need to do is determine whether or not the current joint is the "root" joint. If it is, then we need to record the translation information for this frame as well as the rotation and scale. Start by checking to see if the current attribute is a translation attribute (remember that the "||" symbol means "or"). If so, then we check to see if the current joint (*$each*) is a root joint. We accomplish this by calling the *jcIsRoot* procedure. For now, we are just going to assume that it works. I will talk about the *jcIsRoot* proc after this section. This proc will return a 0 or a 1 depending on if the object passed to it is a root joint or not. (0 for false, and 1 for true). If it is indeed a "root" joint, then we need to write out the values for the current translation attribute. Everything works exactly the same way except for the way we retrieve the value. Since the joints may not be keyed, we need to get the world-space value of the translation another way. We take care of that in the *getBasLoc* procedure which we will talk about in more detail after this section. What this procedure does is return the world space value for the object that is passed in and the attribute that is passed to the procedure. The first argument is the object name, and the second is the attribute that we want the world space value for.

If the joint is not the root joint, then we do not need to worry about recording the translation values and simply use the continue command to skip to the next attribute. Otherwise, if the current attribute is not a translation attribute then we simply store the frame number, get the attribute value from the channel box with *getAttr* and store it all back onto the end of the *$keyVals* string.

```
//fill in "*" for the tangents (since we are keying every frame)
        $inTanTypes = ($inTanTypes + " *");
        $outTanTypes = ($outTanTypes + " *");
        $inTanWeights = ($inTanWeights + " *");
        $outTanWeights = ($outTanWeights + " *");
        $inTanAngles = ($inTanAngles + " *");
        $outTanAngles = ($outTanAngles + " *");

        //record key value
        $inTanTypes = ($inTanTypes + " *");
        $outTanTypes = ($outTanTypes + " *");
        $inTanWeights = ($inTanWeights + " *");
        $outTanWeights = ($outTanWeights + " *");
        $inTanAngles = ($inTanAngles + " *");
        $outTanAngles = ($outTanAngles + " *");

}//do baked
```

Lastly, we fill-in the tangent information with "*" pairs. We do this because the tangents are meaningless when you have a keyframe on every frame of the animation. The reason is that most of the time you are simply going to the next whole frame, and whatever happens in between those frames is never seen. So the curve between the frames is irrelevant.

That's it for the baked joint rotations! All that is left is to organize all of this info into one array and then write the array out to a file.

```
//KeyVal
//check to see if keyvals is empty
if(`size $keyVals` !=0)
{//write it out

  //store info in the preFileArray
  $preFileArray[$preFileCount]=("\nAttribName: " + $thisAttrib);
  //update the counter for each line in the array
  //inc. the counter for each element in the array
  $preFileCount++;
```

We are going to store all of this data into the *$preFileArray*. Once we have stored all of the info there, we can dump each element of the array into a file. Check to see that *$keyVals* is not empty. If it does indeed contain data, we continue on. Basically, this array is going to be an exact copy of what is in the file. We simply write the data, one line at a time, into the array. In addition, we are going to use the *$preFileCount* to keep track of where we are in the array. The first thing into the array is the attribute name that we are currently working with, followed by a colon, then a space, and then the attribute name itself. The attribute name is stored in *$thisAttrib*. After we write each line into the array, we need to increment the *$preFileCount* variable so that we do not overwrite any data.

Start recording all of the actual data for the key:

```
//check if this is the first time through the array
$tempSize = `size $preFileArray[$preFileCount]`;
if($tempSize != 0)
{//make substring

   //if its not he first time through, only copy the new parts in
   //(skip the \nkeyVal: part)
   $tempString = `substring $preFileArray[$preFileCount] 9 ...
                                                        $tempSize`;
}//make substring
else
{//set to ""
  //clear the tempString
  $tempString = "";
}//set to ""

//copy what is in the array and add new items, then store it
//back
$preFileArray[$preFileCount] = ("\nkeyVal:" + $tempString ...
                                                  + $keyVals);
$preFileCount++;
```

If you remember, we are going to start each line with the name of the type of information on each line. We only need this "tag" written once. So what we can do is check the size of the element in an array of strings. Use the size command to get the size of *$preFileArray[$preFileCount]*. If the size of this string is 0 then we know we need to write the "tag". If it is not 0 ,then we simply need to add the next frame/value pair to the end of the string. So we check to see if *$tempSize* equals 0 or not.

If it does then we write the tag, otherwise we write the data out. If the tag is already there, then we are going to use the *substring* command to copy all of the data out of the array element except for the first part that says "nkeyVal:". This is the tag that we are skipping. *Substring* will allow you to store just a section of a string. The fist argument is the input string that we want to get a part of. The second argument is the starting position of the character of the string section and the third argument is the position of the last character in the string that we want. In this case, we want to start at the 9th character in and end with the last character in the string which just so happens to be the size of the string itself. Since we have checked the size earlier, we can just use the *$tempSize* variable that already has the size of the string in it. Once we have a copy of the data that is already in that array element, we then add the "tag" to the beginning of the string and then add to the end the new data which is *$keyVals*. Then we store that whole string back into the array. This overwrits what was previously at that index in the array. If this were the first time through and we did not have the "tag" written, then we simply clear the *$tempString*, write the "tag", then add the *$tempString* which is now blank, and finally add the contents of *$keyVals*. Once we have stored the information back into the array, we increment *$preFileCount* and move on to the next part. We are going to repeat this exact same procedure for all of the data on the tangent as well:

```
$tempSize = `size $preFileArray[$preFileCount]`;

if($tempSize != 0)
{//make substring
  $tempString = `substring $preFileArray[$preFileCount] 13 ...
                                                      $tempSize`;
}//make substring
else
{//set to ""
   $tempString = "";
}//set to ""
$preFileArray[$preFileCount] = ("\ninTanTypes:" + $tempString...
                                                  + $inTanTypes);
$preFileCount++;

$tempSize = `size $preFileArray[$preFileCount]`;
if($tempSize != 0)
{//make substring
        $tempString= `substring $preFileArray[$preFileCount]...
                                                  14 $tempSize`;
}//make substring
else
{//set to ""
        $tempString = "";
}//set to ""

$preFileArray[$preFileCount] = ("\noutTanTypes:" ...
                                    + $tempString + $outTanTypes);
$preFileCount++;

$tempSize = `size $preFileArray[$preFileCount]`;
if($tempSize != 0)
{//make substring
        $tempString= `substring $preFileArray[$preFileCount]...
                                                  15 $tempSize`;
}//make substring
else
```

```
{//set to ""
        $tempString = "";
}//set to ""

$preFileArray[$preFileCount] = ("\ninTanWeights:"
                                  + $tempString + $inTanWeights);
$preFileCount++;

$tempSize = `size $preFileArray[$preFileCount]`;
if($tempSize != 0)
{//make substring
        $tempString= `substring $preFileArray[$preFileCount]...
                                                  16 $tempSize`;
        }//make substring
else
        {//set to ""
                $tempString = "";
        }//set to ""

$preFileArray[$preFileCount] = ("\noutTanWeights:" ...
                                  + $tempString + $outTanWeights);
$preFileCount++;

$tempSize = `size $preFileArray[$preFileCount]`;
if($tempSize != 0)
{//make substring
        $tempString= `substring $preFileArray[$preFileCount]...
                                                  14 $tempSize`;
}//make substring
else
{//set to ""
        $tempString = "";
}//set to ""

$preFileArray[$preFileCount] = ("\ninTanAngles:" ...
                                  + $tempString + $inTanAngles);
$preFileCount++;

$tempSize = `size $preFileArray[$preFileCount]`;
if($tempSize != 0)
{//make substring
        $tempString= `substring $preFileArray[$preFileCount]...
                                                  15 $tempSize`;
}//make substring
else
{//set to ""
        $tempString = "";
}//set to ""

$preFileArray[$preFileCount] = ("\noutTanAngles:" ...
                                  + $tempString + $outTanAngles);
$preFileCount++;
}//write it out
```

Now all we need to do is clear all of the variables and run through the loop again until all of the objects and their attributes have been recorded into the array:

```
//clear list for next attribute
$keyVals = "";
$inTanTypes = "";
```

```
$outTanTypes = "";
$inTanWeights = "";
$outTanWeights = "";
$inTanAngles = "";
$outTanAngles = "";

}//loop through each attribute
}//do each object
}//loop through each frame
```

Now we just write it all out to disk. First up as usual is the header:

```
//write header
print ("All Done! Writing file...\n");
fprint $myFileId "ANIMATION\n";

//Write out value type (absolute or relative)

if($valueType == "bakeKeyValue")
{//do absolute
        fprint $myFileId "valueType: BAKED\n";
}//do absolute
else if ($valueType == "relKeyValue")
{//do relative
        fprint $myFileId "valueType: RELATIVE\n";
}//do relative
else
{//unknown type
        error ($valueType + " is an unknown value type!\n");
}//unknown type
```

Here we simply compare *$valueType* with the two different value type strings "bakeKeyValue" and "relKeyValue". Once there is a match, we write the type out to the file. If the contents of *$valueType* do not equal either of those two, we error out and let the user know.

Now we finally get to write the information to the file:

```
//loop though the array and write each line to the file
for($myCount = 0; $myCount < `size $preFileArray`; $myCount++)
{//write out the array
        fprint $myFileId $preFileArray[$myCount];
}//write out the array
```

To do this, we are again going to use a *for* loop and step through each element of the array. For each element, we simply *fprint* the contents of that element to the file.

```
//close the file
fclose $myFileId;
}//write out animation file
```

Once the info is written to the file, we need to close the file so the data is actually saved to the disk.
Lastly, we close the progress window and end the procedure:

```
        //close progress window
        progressWindow -endProgress;

        }//save the file
}//jcSaveAnimFile
```

That's it! Were done with saving the data! I would suggest making a simple scene and save the data out. Then take a look at the resulting file and try to step through this procedure and follow along in the file. This will help to make sure that you really understand what is going on in the script. All of this information will be used to read the file back into the "Load" tab.

We skipped over a couple of procedures while discussing the file writer. Let's go through them now. First up, we have the *jcIsRoot* proc. As always, we start with the comment block:

```
/*******************************************************
jcIsRoot

Arguments:
        string $myObj - the object that may be a root joint

Returns:
        int - 1 if its a root joint, 0 if not

Notes:
        This procedure will only work correctly on joints.
        Please make sure that only joints are passed to it.
*******************************************************/
```

Just to refresh your memory, this procedure will check to see if the object passed in is a root joint or not. If it is, then it will return a 1, and if not it will return a 0.

```
//see if there are any parents to the current object
$parents = `listRelatives -parent -fullPath $myObj`;
```

The first step to determining if the object is a root joint or not is to check to see if it has any parents. If the joint has no parents then it must be the root of the chain. However, it if does have a parent then we need to check into them a little bit further. So we use the *listRelatives* command with the *-parent* flag and the *-fullPath* flag. This will return all of the parents and store them into the $parents variable.

```
//if there are no parents, then it is the root
if(`size $parents` == 0)
{//return true
        return 1;
}//return true
```

Next we check the size of *$parents*. If the size is 0, then we no there are no parents and the joint is the root. If this is the case, we simply return a "1" to the caller of the procedure.

```
//if there are parents, make sure they are a joint
//if the parent it not a joint, then the joint is the root of
//thatchain

else
{//is it a joint
        for($each in $parents)
        {//check to see if its a joint
                if(`objectType -isType "joint" $each`)
                {//its a joint
                        $isJoint = 1;
                }//its a joint
        }//check to see if its a joint

        if ($isJoint)
        {//not the root

                return 0;

        }//not the root
        else
        {//it is the root
                return 1;
        }//it is the root
}//is it a joint

}//jcIsRoot
```

If the joint does have parents, we need to make sure that they are not joints. If they are joints then the joint that was passed in is not a root joint. However, if the parents are not joints then the joint is a root joint. To do this, we loop through the *$parents* array and use *objectType* to determine if the current parent is a joint. If so, then we set *$isJoint* to be 1. If we never come across a joint in the parent array, then the parents must not be joints and therefore *$isJoint* stays at 0. Next, we check the state of *$isJoint*. If it contains a 1 then the joint is not the root and we return 0. Otherwise it is the root joint and we return 1. These types of procedures are extremely common.

The other procedure that we skipped over earlier is the *getAbsLoc* procedure. As usual, let's start with the comment block and the procedure declaration:

```
/*******************************************
getAbsLoc
Arguments:
        string $myObjName - this is the object name
        string $myAttr - this is the attribute name
return:
        float $absValue - this is the absolute

notes:
        This proc will only work for .translate, .rotate and
.scale
*******************************************/
proc float getAbsLoc(string $myObjName, string $myAttr)
```

As stated before, this procedure is designed to return the world space location of an object, one attribute at a time. We are going to use two arguments, *$myObjName* which is the name of the object in question, and *$myAttr* which is the name of the attribute for which we want the worldspace coordinates. The return value is *$absValue*, which will hold the resulting value for the attribute. This whole procedure basically works off of the xform command in Maya. This command can do many things, however we are going to use it to return to us the translation, rotation, and scale values for an object in world space. To do that we simply put the command into *-query* mode and then use the *-translation*, *-rotation*, or *-scale* flags to tell it which values we want to return. So let's have a look at it shall we?

```
if($myAttr == "translateX" ||
           $myAttr == "translateY" ||
           $myAttr == "translateZ" ||
           $myAttr == "rotateX" ||
           $myAttr == "rotateY" ||
           $myAttr == "rotateZ" ||
           $myAttr == "scaleX" ||
           $myAttr == "scaleY" ||
           $myAttr == "scaleZ")
{//good to go
```

The first thing that we need to do is to check to make sure that the attribute that was passed in is either a translation, rotation, or scale attribute. Otherwise, this procedure will not know how to deal with the attribute since the way we are using xform it will only work for co-ordinates. Check this first by using an *if* statement and comparing the contents of *$myAttr* to the nine different attributes that we are going to handle. Once that is checked, we can move on to getting the location of the object.

```
switch($myAttr)
{//build options

    case "translateX":
    $tempVals = `xform -q -worldSpace -translation $myObjName`;
    $absVal = $tempVals[0];
    break;

    case "translateY":
    $tempVals = `xform -q -worldSpace -translation $myObjName`;
    $absVal = $tempVals[1];
    break;

    case "translateZ":
    $tempVals = `xform -q -worldSpace -translation $myObjName`;
    $absVal = $tempVals[2];
    break;
```

The first thing you will notice here is that we are using a *switch* and *case* statement. This is sort of a shorthand way of writing a bunch of *if-else* statements. Basically, the *switch* command takes one argument. This is a variable that will be compared to each of the case items. If there is a match, then the code in the case statement will be executed. So for instance, if the contents of *$myAttr* was "translateY", then Maya would execute the code for the "translateY" case. Which is this:

```
//Translate Y Case
case "translateY":
$tempVals = `xform -q -worldSpace -translation $myObjName`;
$absVal = $tempVals[1];
break;
```

Notice that each case statement ends with a *break* command. This tells Maya to skip the rest of the *case* statements and then continue to execute code. Now that you understand how *switch/case* statements work, let's take a look into what is happening in each one. This is the "translateX" case for example:

```
//Translate X Case
case "translateX":
$tempVals = `xform -q -worldSpace -translation $myObjName`;
$absVal = $tempVals[0];
break;
```

Inside the case statement, we are going to actually get the world space value of the attribute. So the first thing we do is use xform, as stated earlier, using the *-translation* flag in *-query* mode. We are also using the *-worldSpace* flag to make sure that Maya is returning the "world space" coordinate for the attribute. With these flags, xform will return a float array that contains the X,Y,Z components of the attribute type. Since we are only dealing with one attribute at a time, we only want to return one of the three values. To do this, we are simply going to make another variable called $absVal and store the single value in there. In this example, it would be *$tempVals[0]* which is the index that holds the X value. ([1] holds Y value, and [2] holds Z value). We simply take the contents of $tempVals[0] and store it into *$absVal*. This is exactly the same for all of the cases. The only thing that changes is the *-translation* flag. For rotation and scale, the flag obviously changes to *-rotation* and *-scale* respectively.

The last case is the *default* case. This is the code that gets executed if none of the other cases match the variable used in the *switch* statement. If this happens we simply break.

```
else
{//not a standard location attribute
        error ("getAbsLoc cannot handle the attribute: " ...
                                                + $myAttr + "!\n");
}//not a standard location attribute
```

Lastly, if the attribute passed into the procedure fails the check in the very beginning of the procedure, we drop down to the *else* statement and error out to the user, telling them that *getAbsLoc* cannot handle the attribute passed in. We should never see this in this utility, however if these procedures are used in other scripts, it would be nice to warn the user.

Finally, we simply return the *$absVal* contents to the caller of the procedure and end the proc:

```
return $absVal;
}//getAbsLoc
```

That's all there is to it . Onward to the land of the "Load" tab!

■ **Creating Functionality for the "Load" tab**

Let's start with the "Load Objects From File" button. This is by far and away the most complex button in the whole utility. So might as we get through it first :

```
button
        -label "Load Objects From File"
        -command "$myFileArray = jcLoadAnimFile(\
"loadSourceObjectsTextList\",\"\");\
$myFileType = jcGetFileType($myFileArray);\
$myValueType = jcGetValueType($myFileArray);\
text -edit -label (\"Current Loaded File Type: \" + $myFileType
+ \" \(\" + `tolower $myValueType` + \"\)\") loadedFileTypeText
Label;\
if($myFileType == \"ANIMATION\") {radioButton -edit -
enable true applyToOffset;\ radioButton -edit -enable true
applyToCurrentFrame;\
        intField -edit -enable false loadOffsetFrameField;}\
 else {radioButton -edit -enable false applyToCurrentFrame;\
        radioButton -edit -enable false applyToOffset;\
        intField -edit -enable false loadOffsetFrameField;};\
  if($myValueType == \"BAKED\" && $myFileType == \"ANIMATION\
"){checkBox -edit -value 0 -enable true scaleAdaptCheck;}\
         else {checkBox -edit -value 0 -enable false
scaleAdaptCheck;};"
loadLoadObjectsFromFileButton;
```

Ok, what we have here is a huge mess of a *-command* flag. Normally, I would never recommend writing anything this visually complicated into a button command flag. However, I wanted to keep the procedures from having application specific code in them. In other words, I wanted to keep any of the changes to the UI out of the procedures, and keep them contained within the UI itself. If you look at the commands listed however, it really is not that complicated. The complexity comes from how the *jcLoadAnimFile* procedure works. Let's go through this one line at a time. For right now we are going to skip over the details of how the procedures work, and just get through this button. Then we will get into the procedures themselves.

First we use the *jcLoadAnimFile* procedure to load in a .jca or .jcp file into memory. This is stored into the *$myFileArray* array. Next, we use the *jcGetFileType* procedure to determine if we have loaded an Animation file or a Pose file. Then we use the *jcGetValueType* procedure to determine if we are dealing with *Relative* values or *Joint* rotations. We update the text in the UI to reflect what type of file is loaded and how the values have been stored. We use the text command in *-edit* mode to change the "Current Loaded File Type" text control. Then we start setting the available options based on what kind of file is loaded. So if *$myFileType* is "ANIMATION" then we want to enable the 'applyToOffset' radio button. We also enable the applyToCurrentFrame radio button. Continue on and disable the *loadOffsetFrame* int field since that will only turn on if the "Target Frame Offset" radio button is selected. If *$myFileType* is not, "ANIMATION", then we will disable the *applyToCurrentFrame* and *applyToOffset* radio buttons. We also disable the *applyToOffset* int field, as that will not be needed if we are loading a pose. Lastly, we take a look at the *$myValueType* variable and see if we have a "BAKED" value type loaded along with an "ANIMATION" file type. If this is the case, then we need to enable the *scaleAdaptCheck* check box and uncheck it by default. Otherwise, we make sure the check box is unchecked and then disable it with the *checkBox* command in *-edit* mode. Setting the value to 0 will uncheck the box, where setting the value to 1 will check it. That is really it for the button, so let's move on to the *jcLoadAnimFile* procedure.

```
/*************************************************************

jcLoadAnimFile
Arguments:
        string $myList - this is the name of the textScrollList
to
                         load the object names into
                         (if this is NONE, no list will be
used)
        string $forcePath - if this value is not "" then it
will be
                            used instead of asking for a path
from the user
Returns:
        string[] - returns the poseFileArray.  This contains
the objectNames,
                   attributes, and values from the loaded file.

Notes:
File format for a POSE file is as follows:
Header
valueType: valueType
ObjectName AttributeName AttributeValue <objEnd>
ObjectName AttributeName AttributeValue <objEnd>
....eof

File format for an ANIMATION file is as follows:
Header\n
valueType: valueType

ObjName: nameOfObject\n
AttribName: nameOfAttribute\n
keyVal: frameNumber frameValue\n  (if there is no key for that
frame the values are replaced with a "*")

inTanTypes: frameNumber frameValue\n  (if there is no key for
that frame the values are replaced with a "*")

outTanTypes: frameNumber frameValue\n (if there is no key for
that frame the values are replaced with a "*")

inTanWeights: frameNumber frameValue\n (if there is no key for
that frame the values are replaced with a "*")

outTanWeights: frameNumber frameValue\n (if there is no key for
that frame the values are replaced with a "*")

inTanAngles: frameNumber frameValue\n  (if there is no key for
that frame the values are replaced with a "*")

outTanAngles: frameNumber frameValue\n  (if there is no key for
that frame the values are replaced with a "*")

AttribName: nameOfAttribute\n
...
ObjName: nameOfObject\n
...
...eof
*************************************************************/

global proc string[] jcLoadAnimFile(string $myList, string
$forcePath)
{//jcLoadAnimFile
```

This procedure will load all of the keyframe information saved to a file from the "Save" tab into a big array that we can access when we want to apply the file's information to the current scene. This procedure takes two arguments. The first is the name of the textScrollList to load the object names into. This list should closely resemble the list from the "Save" tab. The second argument is the *$forcePath* argument. Since we will be using this procedure in the "Batch" tab, we do not always want to ask the user to provide a path to the file to be loaded. So this can be used to force our own path and not ask the user. Lastly this procedure will return the array that has all of the contents of the file in it. Also you can see from the notes, how the procedure is expecting the files to be formatted. Notes like those are always good to put into a comment block so a year from now, you can figure out why something isn't working the way that you had expected.

```
//Open a file dialog to find the pose file to load
//Only show the .jcp files in the folder

if($forcePath == "")
{//no forced path
        $myPath = `fileDialog -directoryMask "*.jcp; *.jca"`;
}//no forced path
else
{//path provided
        $myPath = $forcePath;
}//path provided
```

The first thing that we are going to do is check to see if there has been a non-empty path passed into the *$forcePath* argument. If there has, then we need to ask the user for a path, by using the *fileDialog* command. This command is going to take the *-directoryMask* flag as we are going to set that to be "*.jcp" and "*.jca". This says, "open the file browser dialog and then only show files with the .jca or .jcp extensions". This will help make it easier for the user to pick the correct file. After the user opens a file, the path to that file name will be stored into the *$myPath* variable. However, if the path passed in to *$forcePath* is not empty, then we simply take that path and store it into the $myPath variable for later use.

```
//Display to the user
print ("Current Path: " + $myPath + "\n");
//Next we give the user a bit of feedback and tell them what file
//is going to be loaded in.

//if a list was provided, clear the list first
if($myList != "")
{//clear the list
        //clear textScrollList
        textScrollList -edit -removeAll $myList;
}//clear the list

//Here we check to see if there is anything in the
//textScrollList already.  If there is, then we need to remove
//all of those entries before we load our new file in.  Once that
//is don't we can move on.

//Read in the file as long as a path was returned
if($myPath != "")
{//Read File
//Make sure the path is valid
 if(!`filetest -f $myPath`)
 {//bad file
 error($myPath + " was not found, or could not be accessed.\n");
 }//bad file
//Open file for read only and grab id
$myFileId = `fopen $myPath "r"`;
```

Next we are going to make sure that the path that was passed in is not empty. If it is not, then we use the *fileTest* command with the *-f* flag to determine if the file actually exists. If the file does not exist at the provided path, then we error out of the script and tell the user that the file

could not be found. If the file does indeed exist, then we use the *fopen* command with the "r" parameter to open the file in "read only" mode. Once this happens, an internal file ID is assigned to the open file and stored into the *$myFileId* variable so we can access the file later.

```
//Read file into $fileArray[]
//Clear Array to make sure that we start fresh

clear $fileArray;
```

Now we need to prepare the *$fileArray* which will be the main array that holds all of the loaded file's information. Here we simply clear the array with the clear command.

```
//Get header
$myHeader = `fgetword $myFileId`;
print ("myHeader: " + $myHeader + "\n");

//Get valueType
//get descriptor
$myValueType = `fgetword $myFileId`;
print ("myValueType: " + $myValueType + "\n");

//get actual type
$myValueType = `fgetword $myFileId`;
print ("myValueType: " + $myValueType + "\n");
```

Here we finally get to start reading-in some information from the file. To do this, we are going to use the *fgetword* command. This will read in a single word starting from where the current file cursor is sitting. Once the word is read in, the file cursor will be advanced one word. It is important to note that every time you read information from the file, the file cursor position will be updated accordingly. The file cursor is simply a pointer to a place in the file. Its much like the regular text cursor. It is simply the current "read point" in the file. So, if you look in the notes, you will see that the first word in the file is the "header" or the "file type". We read in that word and store it into the *$myHeader* variable. This should be either "ANIMATION" or "POSE". Next, we grab the next word in the same manner. This one should be the same for all files, it should be "valueType:". This is really of no importance to us, but it does help a user looking at the raw saved file to understand what is going on better. What we really want is the word after that. So we again use *fgetword*. This word should be the value type of the file either "BAKED" or "RELATIVE".

```
//check to make sure that $myHeader has something in it
//If it doesn't, its a bad file
if(size($myHeader) == 0 )
{//Bad file
        error "Empty File!";
}//Bad File
```

Now we need to do a bit of sanity checking. We need to make sure that the file actually has something in it, so we can simply check the size of the *$myHeader* variable and make sure that it is not 0. If it is, then we error out of the script and let the user know that the file they tried to open is empty. Now that we know the file is not empty, we can proceed to import it.

```
if($myHeader == "POSE")
{//load a POSE file
```



```
//End the Header line and start reading objects

$fileArray[0] = $myHeader;
$fileArray[1] = $myValueType;
```

The next thing we need to decide is which type of file is loaded and import it accordingly. Check the *$myHeader* variable. If it is equal to "POSE" then we work on opening the "Pose" file format. Once it is established that we are working with a "POSE" file, we can put the *$myHeader* value into the first index of the *$fileArray*. Then we can put the *$myValueType* into the second index.

```
//get first objectName
$objString = `fgetword $myFileId`;
```

If we check our notes again, we will see that the next item in the file is the name of an object. Let's store that into the *$objString* variable to use later on.

```
//Add it to the list

if($myList != "")
{//do not edit list
        textScrollList -e -append $objString $myList;
}//do not edit list
```

Now that we have the name of the object, we can append the name of the object to the "Source Objects" list. First, however, we check to see if the *$myList* argument that was passed-in actually contains a name of a textScrollLlist. If it does not, then we do not want to actually add the object name to the end of any list.

```
//Loop through each word in the file until "endOfFile" is reached
while(!`feof $myFileId`)
{//Store All file info into array
        //get next word from file
        $word = `fgetword $myFileId`;
```

Now we can start reading in the rest of the line until we hit the "*objEnd*" tag at the end of the line. This is where we use a *while loop* to continue to read until the *feof* is reached. This is a built in command that checks to see if the "End Of File" has been reached. If it has, the *while loop* stops reading in items. Assuming that we have not reached the end of the file, we will read in the next word which will be the next attribute name for the object that we have information for.

```
if($word != "<objEnd>")
{//Save attributes
        $objString = ($objString + " " + $word);
}//Save attributes
```

Now that we have the next word, we want to make sure that it is not the "objEnd". This would mean that we have read everything there is to read in for the current object and we will need to move on to the next object. If we have not hit the end of the object, then we need to take the current word, and tack it on to the end of the *$objString* string with a space in between. What we are going to do is make one long string out of each line of the file. Then we will store that entire sting into one index of the *$fileArray*. Basically, we keep adding the words onto the line until we hit the "objEnd" word.

```
else
{//Next Object
        //store finished object in array
        $fileArray[`size $fileArray`] = $objString;

        //Get next object name
        $word = `fgetword $myFileId`;
```

OK, so say we did hit the end of the line. The first thing we do is store the finished line into the *$fileArray* variable. Now that we are sitting on the "objEnd" word, we know that the next word in the file will be the name of the next object. Knowing this, we simply use *fgetword* and store it into the *$word* variable.

```
if($myList != "")
{//do not edit list
        //add object to textScrollList
        textScrollList -e -append $word $myList;
}//do not edit list
```

We check to see if a textScrollList has been passed in. If it has, then we need add the next object name to the list. Just like before, we use the *textScrollList* command in *-edit* mode and append the name.

```
                //clear string
                $objString = "";

                //add new objectName to the new objString
                $objString = $word;

        }//Next Object
}//Store all file info into array
```

Now that we have our new object, it is time to start a new long string for the next object. Clear *$objString* by setting it to nothing (""). Then we set *$objString* to equal *$word*. This will have the effect of initializing the new string with the next object name in the file. Then we simply close-off the loops.

```
        //We are now done with the file.  Close it.
        fclose $myFileId;
}//Load a POSE file
```

Once we have read-in all of the information from the "POSE" file, we can close the file again.

That pretty much does it for the "POSE" files, so let's move on to the "ANIMATION" type files.

```
else if($myHeader == "ANIMATION")
{//Load animation file
        print ("Loading Animation...\n");
        //load header
        $fileArray[0] = $myHeader;

        //load valueType
        $fileArray[1] = $myValueType;
```

If the $myHeader variable contains "ANIMATION", then we need to start loading in the "ANIMATION" data. Firstly, we are going to give the user a bit of feedback and let them know that we are starting to load an animation. Just as before, we save-off the $myHeader value and the $myValueType value into the $fileArray for later use.

```
//read each line of the file.  If its an object name, add it to
//the list
$animFileCount = 2;
```

It's time to set-up a counter. We will use this to start on the third index in the array. With each added line to the array, we will increment this by 1.

```
while(!`feof $myFileId`)
{//read in animation file
        //read in each line
        $myTempString = `fgetline $myFileId`;
```

Just like with the "POSE" files, we are going to continue to read each line of the file in until we hit the EOF (end of file). So again, we are going to use a *while loop*. Now we will read in each line, one line at a time. To do this, we are going to use *fgetline* instead of *fgetword*. This time, the *$myTempString* will hold one entire line of the file.

```
//break it down and check for "ObjName"
$numTokens = tokenize($myTempString, $tempBuffer);
```

Next we are going to break up that string with the *tokenize* command and store the resulting tokens into the *$tempBuffer* array.

```
//check to see if its an Object Name line
if($tempBuffer[0] == "ObjName:")
{//store object name

        //Update the list
        if($myList != "")
        {//edit list
           textScrollList -edit -append $tempBuffer[1] $myList;
        }//edit list

//update the count to the next object name
//we will use this count later on to do custom mapping
$fileArray[0] = ($fileArray[0]  + " " + $animFileCount);

}//store object name
```

Now take that *$tempBuffer* variable and check to see if the first item in it is "ObjName:". If it is, then we know the next item is the name of the object. Assuming that we have passed-in a name of a textScrollList, we can use the *textScrollList* command in *-edit* mode to append the object name to the end of the list. The next part is a little bit tricky. To save time later on when we are trying to apply all of the information to the scene, we are going to store the index number of the object name in the *$fileArray*. This way we can skip through the array and look for all of the object names without having to actually "search" through the array to find them. We are going to store this data on the first element of the *$fileArray*. Right now, that index also contains the file type name. So, we are just going to stick the index numbers of the object names on after the file time name in the first element of the array. The first element may look like this at some point:

```
"ANIMATION 2 18 34 90"
```

Where "ANIMATION" is the file type and the numbers are the index numbers where we can find the names of all of the objects in the array.

```
//store the info into the array
//Strip off newLine
tokenize($myTempString, "\n", $tempBuffer);
```

Now that we have read-in the next line, we need to get rid of the "\n" character at the end of the line. This is of no use to us now, but it sure does make reading the file in much easier. The box character defines the end of a line, and the *fgetline* command looks for those when reading in a line.

```
//store into fileArray
$fileArray[`size $fileArray`] = $tempBuffer[0];
```

Now we want to actually store the information into the *$fileArray*. *$tempBuffer[0]* now contains the line we want without the "n" character at the end of it.

```
//Inc. the counter
$animFileCount++;
```

Increment the counter and run through the loop as many times as needed.

```
        }//read in animation
                print "Done...\n";
        }//Load animation file
}//Read File
```

Now that we have finished, we simply let the user know that we are done, and close-off all of the loops.

```
//return the fileArray to the caller
return $fileArray;
```

Lastly, we return the new *$fileArray* to the caller of the procedure, which in this case was the button from the "Load" tab.

We have skipped over a couple of procedures that we called within the *jcLoadAnimFile* procedure. Those were the *jcGetValueType* and the *jcGetFileType* procedures. Let's go over them now.

```
//*****************************************************************
jcGetValueType
Arguments: string $myFileInfo[] - the array the holds the
currently loaded file info

Returns: string - returns the first word in $myFileInfo[1]
                  (usually "BAKED" or "RELATIVE"

Notes:
        none
*****************************************************************
*/
global proc string jcGetValueType(string $myFileInfo[])
{//jcGetValueType
```

This procedure will return the "Value Type" of the loaded file. It only takes one argument, and that is the array that is holding all of the loaded file info.

```
//split up the first line of the animation info
$numTokens = tokenize($myFileInfo[1], $tempBuffer);

//Grab the first word and check it
return $tempBuffer[0];
```

The way it works is very simple. We simply tokenize the contents of the second index of the file array (*$myFileInfo[1]*) and return the first word in that string.

```
}//jcGetValueType
```

Then we close the procedure. Not much to it really, but it saves a lot of typing later on for us.

The *jcGetFileType* procedure works exactly the same way:

```
//*****************************************************************
jcGetFileType()
Arguments: string $myFileInfo[] - The array that holds the
currently loaded file info

Returns: string - this returns the first word in $myFileInfo[0]
                  Usually "ANIMATION" or "POSE"

Notes:
        This is simply a handy little proc to grab the file type
*****************************************************************
*/
global proc string jcGetFileType(string $myFileInfo[])
{//jcGetFileType
        //This takes the exact same arguments.

        //split up the first line of the animation info
        $numTokens = tokenize($myFileInfo[0], $tempBuffer);

        //Grab the first word and check it
        return $tempBuffer[0];

}//jcGetFileType
```

The only difference is the index that we pull the information from. The "file type" string is the very first index in the array, *$myFileInfo[0]*. We grab that and then tokenize it so we can take the first element from the *$tempBuffer* array. This is an important step because if you remember, we will also be storing the index numbers of the object names in that element as well. We want to make sure that we only grab the first word in the string, which is the "file type".

Let's move on to the other buttons in the first section of the "Load" tab. Those are the Add and Remove Target buttons. This will allow the user to add "targets" for each object in the source file. This means that any animation information for the source object will get copied to the "targets". This is useful if the target character is different from the source character or if the controls for the rig are named differently in the scene. We want this to work by selecting the "Source" object in the "Source List" and then adding targets (or removing targets from) that object. When we select another object, the "Target" list is updated to show only the targets associated with the currently selected source object. Let's take a look at the buttons themselves:

```
button
        -label "Add Targets"
        -command "$myTargetArray = jcAddTargets(`ls -sl`,
        $myTargetArray,
        \"loadSourceObjectsTextList\",
        \"loadTargetObjectsTextList\");
        \jcUpdateTargetList(\"loadSourceObjectsTextList\",
        \"loadTargetObjectsTextList\",
        $myTargetArray);"
        loadAddObjectButton;


button
        -label "Remove Targets"
        -command
        "$myTargetArray=jcRemoveTargets($myTargetArray,\
        "loadSourceObjectsTextList\",
        \"loadTargetObjectsTextList\");
        \jcUpdateTargetList(\"loadSourceObjectsTextList\",
        \"loadTargetObjectsTextList\",
        $myTargetArray);"
        loadRemoveObjectButton;
```

The two are nearly identical, except one calls *jcAddTargets* and the other calls *jcRemoveTargets*. Once those procedures have finished, we then call jcUpdateTargetList to refresh the target list. First lets start with *jcAddTargets*:

```
//*****************************************************************
jcAddTargets()

Arguments:
        string $myNewTargets[] - list of names of objects to
add to target list
        string $myMappingArray[] - the array that holds the
names of the targets
        string $mySourceList - name of the textScrollList that
holds the source object names
        string $myTargetList - name of the textScrollList to
add names to

Returns:
        string[] - returns the updated mapping array

Notes:
        none
*****************************************************************/


global proc string[] jcAddTargets(string $myNewTargets[], string
$myMappingArray[], string $mySourceList, string $myTargetList)
{//jcAddTargets
```

This procedure takes four arguments. The first is the *$myNewTargets* array. This is simply the list of names that we want to add as targets. If you look at the button code, you will see that we pass in "ls –sl" as the argument. This basically passes in the list of selected items and because it is between back quotes, it is executed first, then passed to the procedure. Secondly, we have the *$myMappingArray*. This is an array that holds all of the target names for all of the objects. This array is setup so that each index in the array corresponds to each item in the "Source Objects" textScrollList. Each element in the *$myNewTargets* array has all of the targets for the object that it corresponds with. Each target name is separated by a space. So we may have something like this for example:

```
$myTargetArray[4] == "pCube1 pCone2 l_handCTRL";
```

The third argument is the name of the textScrollList that holds all of the source object names, and lastly is the name of the textScrollList where the target names are to be stored. Once the procedure has completed it will return an array of strings that hold the updated "mapping" array.

```
//get selected object index

$myIndex = `textScrollList -query -selectIndexedItem ...
                                                $mySourceList`;
```

When the procedure is called, the first thing we need to do is determine which "Source Object" is actually selected so we can determine which targets to show. We do this by using the *textScrollList* command to query the selected item's index in the list. For this, we use the *–selectIndexedItem* flag, which will return the "slot number" of the item that is selected in the textScrollList.

```
//make sure we have a source selected
if(`size $myIndex` != 0)
{//source selected
```

Next we want to make sure that there is an object actually selected, so we can simply check the size of the *$myIndex* array. If it is not 0, then we have an item selected.

```
//loop through list of selected objects
for($each in $myNewTargets)
{//add new targets
        $tempList = $myMappingArray[$myIndex[0]];
```

The next thing we need to do is determine if the target has already been loaded into the list. In order to do this, we need to loop through each of the selected objects in the scene that are to be added into the list. Once we start the loop, we need to load-in the targets that are already assigned to the source object. These, of course, are stored in the *$myMappingArray* array at the index of the selected "Source Object". Once we read-in that whole list (which if you remember is a long string with each target object name separated by a space) we need to tokenize all of the target names into an array. As usual, we use the *tokenize* command to accomplish this.

```
//split up
$numTokens = tokenize($tempList, $tempBuffer);
```

Once we have this, we then need to check for duplicates:

```
//check for duplicates
for($item in $tempBuffer)
        {//look for dupes
                if($item == $each)
                        {//its a dupe
                                $isDupe = 1;
```

```
                    }//its a dupe
        }//look for dupes
```

We again are going to use a *for-in* loop to step through each of the objects in the array. Then we simply compare the currently selected object in the scene, the one that we are trying to add, with each object in the current target list, which is now stored in *$tempBuffer*. If there is a match, then we set the *$isDupe* variable to be 1.

```
        if($isDupe == 0)
        {//unique
                $tempList = ($tempList + " " + $each);
                $myMappingArray[$myIndex[0]] = $tempList;
        }//unique
        else
        {//reset
                $isDupe = 0;
        }//reset
    }//add new targets
}//source selected
```

Finally we check to see if *$isDupe* has been set to 1. If it has not, then we know we can safely add the new target. To actually achieve this, we simply take the *$tempList* variable, which is holding the long string with all of the target names in it, and add a space and then the new target name right on to the end of the list. Once the *$tempList* variable has been updated, we stick it back into the mapping array, replacing the mapping list at for the currently selected "Source Object". If *$isDupe* is not set to 0, then we do nothing with the target list, and reset $isDupe to 0, for the next go around the loop.

```
//return the updated custom mapping array
return $myMappingArray;

}//jcAddTargets
```

Lastly, we return the new mapping array and close off the procedure.

Now, if you will notice, we never did anything to the "Target Objects" textScrollList. So, since Maya will not actually do anything, that we don't tell it to do, we need to update that list so the user can see what is going on. For this, we will used *jcUpdateTargetList*. This procedure will need to be called every time we either add or remove items from the "Target Objects" textScrollList.

```
/*************************************************************

jcUpdateTargetList()

Arguments:
        string $mySourceList - this is the name of the Source
Objects List
        string $myTargetList - this is the name of the Target
Objects List
        string $myMappingArray[] - this is the array that holds
the targets
                                   for the source objects
Returns:
        none

Notes:
```

This will fill the custom mapping Target List with the target objects assigned to the currently selected Source Object

```
*************************************************************/

global proc jcUpdateTargetList(string $mySourceList, string
$myTargetList, string $myMappingArray[])
{//jcUpdateTargetList
```

This procedure also takes three arguments. First is the *$mySourceList*. This is the name of the "Source Objects" textScrollList. Second, we need to pass-in the name of the "Target Objects" textScrollList. Lastly, we need to pass in the *$myMappingArray*, which is the array that holds all of the target lists. There is actually not much to this, so lets just jump right in.

```
//clear target list
textScrollList -edit -removeAll $myTargetList;
```

The very first thing that we need to do is clear whatever is in the "Target Objects" textScrollList. For this we simply use the *textScrollList* command in *-edit* mode and use the *-removeAll* flag.

```
//get index of selected object
$myIndex = `textScrollList -query -selectIndexedItem ...
                                                $mySourceList`;
```

Now we need to get the "slot number" of the selected item in the "Source Objects" list. This works exactly the same as it did in the *jcAddTargets* procedure.

```
//Break apart target list for selected object

$numTokens = tokenize($myMappingArray[$myIndex[0]], $myTokens);
```

Once we have that index number, we can grab the target list from the *$myMappingArray*, remember, this is just a long string of target names separated by spaces. Once we have that string, we need to break it up into an array, so again we use the *tokenize* command and store the tokens into the *$myTokens* array.

```
//make sure there are targets listed

if($numTokens > 0 && $myTokens[0] != "")
{//update list
        for($each in $myTokens)
        {//fill in list
                textScrollList -edit -append $each $myTargetList;
        }//fill in list
}//updateList
```

Now that the list is broken apart, we need to check to make sure that there were indeed targets already assigned to this particular source object. We can check this by making sure the number of tokens is greater than 0, and that the first item in the list is not blank. Once we are sure that we have some targets to display, we loop through each of the items in the *$myTokens* array, and use the *textScrollList* command to append each new item name to the end of the target list.


```
else

{//nothing to update
        textScrollList -edit -append "No Targets" $myTargetList;
}//nothing to update
```

If there are no targets assigned as of yet, then we simply put the words "No Targets" into the "Target Objects" list.

```
}//jcUpdateTargetList
```

Lastly, as always, we close the procedure.

Now that we can add to and refresh the target list, we need to be able to remove those targets as well. To do that, we will call upon the *jcRemoveTargets* procedure.

```
/*************************************************************
jcRemoveTargets()

Arguments:
        string $myMappingArray - the global array that holds
the mapping info
        string $mySourceList - name of the source objects
textScrollList
        string $myTargetLIst - name of the target objects
textScrollList

Returns:
        the new mapping array

Notes:
        none
*************************************************************/


global proc string[] jcRemoveTargets(string $myMappingArray[],
string $mySourceList, string $myTargetList)
{//jcRemoveTargets
```

This procedure takes three arguments, just like the *jcAddTargets* procedure. The first is the *$myMappingArray* argument, which is simply the array that is holding all of the target lists. The $mySourceList and *$myTargetList* arguments are the names of the "Source Objects" textScrollList, as well as the "Target Objects" textScrollList.

```
//Get the proper index to use from the $mySourceList
textScrollList
$myIndex = `textScrollList -query -selectIndexedItem ...
                                                $mySourceList`;
```

First we need to get the "slot number" of the selected item in the textScrollList. We can then use this to determine which target list goes with that object.

```
//read in the targets
$tempList = $myMappingArray[$myIndex[0]];
```

Here we read in the long string that is the target list for the selected item and store that into the *$tempList* string.

```
//Break down list into parts we can search

$numTokens = tokenize($myMappingArray[$myIndex[0]], $myTargets);
```

Now we need to break apart that string into the *$myTargets* array. This array now only holds the names of the targets for the selected object.

```
//grab list of selected objects to delete

$deleteMe = `textScrollList -query -selectItem $myTargetList`;
```

Next, we need to determine which of the items in the "Target Objects" list are selected. Again, we can simply use the *textScrollList* command in *-query* mode and use the *-selectItem* flag to return the names of the selected items.

```
//match up the strings and replace the deleted ones with "*DEL*"
for($each in $deleteMe)
{//loop through deleteMe
        for($i = 0; $i < `size($myTargets)`; $i++)
        {//loop through mapping array
                if($each == $myTargets[$i])
                {//flag for deletion

                $myTargets[$i] = "*DEL*";
                }//flag for deletion
        }//loop through mapping array
```

Next, we will loop through each item in the *$deleteMe* array and compare it with each item in the $myTargets array. We do that by looping through the *$myTargets* array and doing a compare between $each which is the current item to be deleted, and *$myTargets[$i],* which is the entire list of targets for the selected source object. When there is a match, we will replace the name of the deleted target in the *$myTargets* array with "*DEL*". This way all we have to do is copy the array back and skip any elements that have "*DEL*" in them.

```
//make new array and leave out the deleted ones
$myMappingArray[$myIndex[0]] = "";
```

Once all of the proper items have been marked for deletion, we can clear the element in the array that was holding the target list for the selected source object.

```
//make new array
for($i = 0; $i < `size($myTargets)`; $i++)
{//make clean array
  if($myTargets[$i] != "*DEL*")
  {//write target back
   $myMappingArray[$myIndex[0]]=($myMappingArray[$myIndex[0]]...
                                        + " " + $myTargets[$i]);
  }//write target back
}//make clean array
```

All that is left to do now, is loop through the *$myTargets* array and copy back the elements in the array that do not match "*DEL*". So all we do is set up a for loop and inside that, check to see if *$myTargets[$i]* equals "*DEL*" or not. If it does not, then we take what is in *$myMappingArray[$myIndex[o]]* and add to it the next target that has not been deleted, *$myTargets[$i]*. If the next target has been deleted then we simply skip it and move on to the next one.

```
}//loop through deleteMe
return $myMappingArray;
}//jcRemoveTargets
```

Now all we have to do is close the outer loop, return the new *$myMappingArray* and close the procedure.

At this point, we can load an animation in, add targets, and remove targets. Now let's move on to the most important, and most complex three procedures, *jcApplyAnim*, *jcApplyKeys*, and *jcApplyAttribs*. We will use these three procedures to apply the loaded animation to the scene.

```
/*****************************************************************

jcApplyAnim()

Arguments:
string $myAnimInfo[] - this is the array that holds the
animation info from the file to be parsed

int $autoLoad - 0 or 1,   0 == Auto map the objects
                          1 == use custom mapping

string $onError - what to do if part of the auto load fails.
This has no effect if $autoLoad == 0

int $doScale - 0 or 1, Whether to adjust for scale or not.

string $offsetType - how to handle the offset:
"applyToCurrentFrame" or "applyToOffset"

int $myOffset - how many frames to shift the animation before
applying it.

Returns:
        none

Notes:
This proc does the actual "paste" of the key info to the targets
*****************************************************************/



global proc jcApplyAnim(string $myAnimInfo[], string
$myTargets[], int $autoLoad, string $onError, int $doScale,
string $offsetType, int $myOffset)

{//jcApplyAnim
```

This procedure will be called directly from the "Copy Source to Targets" button and takes a bunch of arguments, most of which are simply user set options from the UI. The first one is the *$myAnimInfo array*. This is the array that holds the file that we read-in with the "Load Objects From File" button. The second argument is the *$autoLoad* option. This is going to be used to either automatically match up the items in the "Source Objects" list, or use the custom targets to apply the animation to. The next one is the *$onError* argument. This tells the script what to do in the case of an error when automatically mapping objects to the targets. The fourth argument is the *$doScale* argument. This tells the script to compensate for a size difference between the joint structure of the source objects and the target objects. Next we have the *$offsetType* argument which tells the script how to hand the frame offset. Should we use the frame numbers stored in the .jca file or start the animation on a custom frame. Lastly, we have the *$myOffset* argument. If we are using a custom offset to start the animation, then this is the argument that tells the script how many frames to shift the animation.

```
print "Starting Copy...\n";
        //open progress window
        progressWindow
                -title "Applying Frames"
                -progress $progressAmount
                -status "Setting Pose: 0%"
                -isInterruptable true;
```

First we are going to give the user a little feedback, by telling them that we are starting to copy the animation to the scene. Then we open up a progress window to show them how far along we are.

From here on in we are going to break this procedure up into two halves, one that copies a pose, and one that copies the animation. Let's start with the pose.

```
//Figure out which type of file this is:
if(jcGetFileType($myAnimInfo) == "POSE")
{//POSE file
```

Get the file type of the file that we have loaded into memory. If the file type is of type "POSE", then we continue:

```
//loop through array and apply pose
//start at 2 because 0 is the header and 1 is the valueType
for($i = 2; $i < `size($myAnimInfo)`; $i++)
{//step through objects
```

Here we are going to start at the third element of the array, (just past the header) and start looking at each object that is in the array.

```
//update progress
$precent = (((float)$i)/(`size $myAnimInfo`));
$progressAmount = ($percent * 100);

//update progress window
if(`progressWindow -query -isCancelled`)
    {//warning
        warning ("Not all keys were set! \(user cancelled\)\n");
                break;
    }//warning

progressWindow
        -edit
        -progress $progressAmount
        -status ("Setting Pose: " + $progressAmount + "%");
```

Since we are at the top of the loop, let's start the progress window. This is going to work exactly the same as it did in the "Save" tab. We take our counter, *$i*, and divide it by the total number of times we go through the loop. This will compute a percentage of how much we have done. We then simply take that percentage, and multiply it by 100 to move the decimal out of the way. After that, we update the progress window with the new information.

```
//read in each line of array (each object) and tokenize
$numTokens = tokenize($myAnimInfo[$i], $myCurrentObject);
```

Now we tokenize one element of the *$myAnimInfo* array at time and store it into the *$myCurrentObject* array. If you remember the format for the POSE file, you will remember that each object has all of it's information in one element of the array. The string starts with the object name, then the attribute name, then the attribute value. These will now be all nicely split up for us in the *$myCurrentObject* array.

```
//set up source object
$mySource = $myCurrentObject[0];
```

We must setup the source file in the case that the user has selected "Auto Mapping". All we are really doing is taking the object name and storing into the *$mySource* string for easy access.

```
if($autoLoad == 1)
{//autoLoad file
  //setup target object
  $targets[0] = $myCurrentObject[0];
  $myTarget = $myCurrentObject[0];
```

Here we will check to see if *$autoLoad* has been turned on. If it has, then we must setup the target object to be the same as the source object This is done before actually checking for the existence of the object.

```
//check for object
$isObject = `objExists $myTarget`;
```

Now that we have the name of our target object, we need to check to see if the object actually exists before trying to apply the pose to it. For this, we can use the *objectExists* command. This will simply return a 1 or 0 if the object does or does not exist.

```
//Take proper action based on $onError result
if($isObject)
{//object exists, apply the attributes
        jcApplyAttribs($i, $myAnimInfo, $targets);
}//object exists, apply the attributes
```

Here we determine what the result of the *objectExists* command was and decide what to do with it. If the object does indeed exist, then we use the *jcApplyAttribs* command to set the pose for that object. We will cover that procedure in much greater detail in a little while.

```
else if($onError == "onErrorContinue")
{//skip this one and move on
        continue;
}//skip this one and move on

else if($onError == "onErrorCancel")
{//stop on error
        error ($mySource + " does not seem to exist, ...
                                application stopped\n");

}//stop on error
```

If the object did not exist when we checked, then we need to check the arguments to see what the user wanted to do if there was an error. If the user had the utility set to continue if there is an error, then we simply use the *continue* command to skip the rest of the loop and move on to the next object. However, if the user has the utility set to cancel in the case of an error, then we error-out of the script and let the user know that the object did not exist.

```
else if($onError == "onErrorAsk")
{//ask to continue or move on

    $onErrorDialog = `confirmDialog -title "Error Finding Object"
                -message ($mySource + " was not found.  What...
                                would you like to do?")
                -button "Continue"
                -button "Cancel"
                -dismissString "Cancel"
                -defaultButton "Continue"`;


    //process response
    if($onErrorDialog == "Continue")
    {//do next object
        continue;
    }//do next object
    else
    {//stop
        error "User Cancelled Operation";
    }//stop
}//ask to continue or move on
```

The last option that a user may set is to have the utility ask what to do next. Basically we want to know if we should continue to process the pose or stop. We are going to use the *confirmDialog* command to do this. This command will open up a dialog for the user with a message and a number of buttons. We can set the message with the *-message* flag. Then we can setup the buttons with the *-button* flag. In this case, we will have two buttons, one being "Continue", and the other being "Cancel". The *-dismissString* flag, sets the string that the dialog will return if the dialog box is closed with the close button on the window. The *-defaultButton* flag, sets up which button is automatically highlighted when the dialog box is first displayed. Any response that the dialog box will give us back will be stored in the *$onErrorDialog* variable.

Once the dialog is dismissed, we can deal with the user response. First we will check to see if the *$onErrorDialog* contains the string "Continue". If it does, that means that the user pressed the "Continue" button. The string that the *confirmDialog* command returns is always the text on the button, or the text passed in with the *-dismissString* flag. If the user did choose continue, then we simply use the *continue* command to skip to the next object in the loop. Otherwise, we stop the script with an error and report that the user cancelled the operation.

```
else
{//Use custom mapping
        //get list of targets and tokenize
        $numTokens = tokenize($myTargets[$i-1], $targets);
        jcApplyAttribs($i, $myAnimInfo, $targets);

}//use custom mapping
```

This *else* goes with the first *if* statement that checked to see if *$autoLoad* was set to 1. If it is not, then that means the user wishes to set custom mappings for the target objects. In this instance, we need to tokenize the target list for the current "Source Object" and pass in the resulting array to the *jcApplyAttribs* procedure along with the current count, and the *$myAnimInfo* array. Also, if you are wondering why we are using the -1 part in the *$myTargets[$i-1]*, it is because if you remember, we started on *$i* being 2 in order to skip the header, so we need to subtract 1 from the count to get the correct target in the *$myTargets* array. I am sure you are wondering why we are only subtracting 1 since arrays start at 0. Well, if you remember, we skipped the first element in the $myTargets array because we needed to keep the index of that array consistent with the "slot number" of the source object in the "Source Objects" array.

The next part of this procedure will get into applying an animation file, so let's stop here and go over the *jcApplyAttribs*, which is where all of the real magic is happening.

```
/*************************************************************
**************
jcApplyAttribs()

Arguments:
        int $mySourceIndex - index of Source object
        string $mySourceArray[] - the main info array that
holds all of the source info
        string $myTargets[] - array that holds the target names
for this object

Returns:
        none

Notes:

*************************************************************
**************/
global proc jcApplyAttribs(int $mySourceIndex, string
$mySourceArray[], string $myTargets[])
{//jcApplyAttribs
```

This procedure is going to actually do the "setting" of the pose itself. We are going to pass to it three arguments. The first being *$mySourceIndex* which is the index of the source object in the *$mySourceArray* array. The second argument is the array that is currently holding all of the loaded-in file data. Lastly, we pass-in the array that holds the names of the targets for the current source object.

```
//Get ValueType
$myValType = $mySourceArray[1];
```

First we need to figure out what types of values we are working with. They are either the "Baked" joint values, or "Relative" values. The result is stored into the *$myValType* string.

```
//break down source and attributes
$numTokens = tokenize($mySourceArray[$mySourceIndex], ...
                                                $currentObj);
```

We need to breakdown all of the items in the long string that contains the object name and all of it's attributes. This will get stored into the *$currentObj* array.

```
//loop through each target
for($each in $myTargets)
{//Apply attributes to targets
//From here we will loop through each of the targets and set all
//of the attributes for them.

//check that the object has at lease one target
if($each == "")
{//no targets.. skip
warning ($currentObj[0]+" does not seem to have any targets! ...
                                                Skipping...\n");
continue;
}//no targets.. skip
else if(!`objExists $each`)
{//target doesn't exist.. skip

warning ($each + " does not seem to exist! Skipping...\n");
continue;
}//target doesn't exist.. skip
```

We need to do a bit of sanity checking. First, we make sure that *$each*, which is the current target, actually has something in it. If it does not, then we warn the user that the current source object, does not actually have any targets, and then skip to the next object.

If *$each* does have something in it, then we need to make sure that the target really does exist. Again, we will use the *objExists* command to make sure that the target actually does exist. If it does not, then we warn the user, and skip to the next target.

```
else
{//apply

  for($i = 1; $i < `size($currentObj)`; $i+=2)
  {//loop through each attribute ($i should be at the attr name)
        //check that attribute exists

        clear $attribCheck;
        $attribCheck = `listAttr -scalar -write -string ...
                                        $currentObj[$i] $each`;
```

If everything checks out, we can try to apply the attribute. So we setup a loop to start applying the values to the targets. First we need to check to make sure that the attribute actually exists on the object. We are going to use the *listAttr* command to check if the attribute exists. We will check that the attribute is writable, that it is a scalar value, and that it matches the name that is in the *$currentObj[$i]* element. If it does exist, then *$attribCheck* should have at least one element in it.

```
//if the attribute is there then apply, else warn about it, but
//continue
if(`size $attribCheck` == 0)
{//attrib not found
        warning ("Can't write to " + $each + "." + ...
                        $currentObj[$i] + "... skipping...\n");
}//attrib not found
else
{//apply attribs
        eval("setAttr " + $each + "." + $currentObj[$i] + " " +
                        $currentObj[$i+1]);

                }//apply attribs
          }//loop through each attribute ($i is the attr name)
      }//apply
   }//Apply attributes to targets
}//jcApplyAttribs
```

This is where we check that the size of *$attribCheck* is more than 0. If it is not, then we warn the user. Otherwise, we use the *setAttr* command on the *object.attributeName* and give it the value stored in *$currentObj*, one element after the attribute name. This is why we use *$currentObj[$i+1]* to get the value of the attribute that we need to set. That about does it for the "POSE" file format. Let's move on to the "ANIMATION" format.

```
else if (jcGetFileType($myAnimInfo) == "ANIMATION")
{//Animation file

        //Get objectName index
        tokenize($myAnimInfo[0], $indexList);

        //add final index to index list
        $indexList[(`size $indexList`)] = `size $myAnimInfo`;
```

If the file type is "ANIMATION", then we will execute this part of the script. Tokenize the first element in the *$myAnimInfo* array. If you remember, we will be using this element in the array to not only hold the file type, but also the index values in the *$myAnimInfo* array that are the start of a new object. In other words, these are the indices of the array that hold the "ObjName" tag. We can now refer to those index values to find the next object instead of searching the entire array each time. To make this a bit easier on ourselves, we will store the index values into their own array called *$indexList*. This is what the *tokenize* line does for us.

Next we add the last index value of the *$myAnimInfo* array to the index list so we know when we have hit the last element in the array.

```
//step through each object
for($i = 1; $i < ((`size $indexList`)-1); $i++)
{//do each object
        //Determine Progress Amount
        $percent = (((float)$i)/(`size $indexList`));
        $progressAmount = ($percent * 100);

        //update progress window
        if(`progressWindow -query -isCancelled`)
        {//warning
        warning("Not all keys were set! \(user cancelled\)\n");
        break;
        }//warning

        progressWindow
                -edit
                -progress $progressAmount
                -status ("Applying Animation: " + ...
                                        $progressAmount + "%");
```

Now we can setup the main loop that will step through each of the objects using the index list in the *$indexList* array. Once the loop is started, we can now also setup the progress window to give the user a bit of feedback. Here we are using *$i* as the counter and the size of the *$indexList* as our total. Once we do the divide, we get our percent, then we simply multiply by 100, to make it a more legible number. Once that is done, we update the progress window accordingly.

```
//store the current index
        $currentIndex = $indexList[$i];
```

Now we are going to store the current object's index into a variable so we can easily access the data when we want to. This will be stored into the *$currentIndex* variable.

```
//clear $mySource
        $mySource = "";
```

Here we simply clear out anything that may have been left over in the *$mySource* variable from a previous run.

```
//setup source object name

        tokenize ($myAnimInfo[$currentIndex], $tempBuffer);
        $mySource = $tempBuffer[1];
```

Once that has been cleared out, we can fill it with the current source object's name.

```
//setup targets
        if($autoLoad ==1)
                {//autoload file
                        //target same as source
                        clear $targets;
                        $targets[0] = $mySource;
                        $myTarget = $mySource;
```

Now we can setup what targets the source information is going to be applied to. This part will work almost exactly the same way that it did for the "POSE" file format. We check to see if the automatic mapping has been turned on. If it has, then we clear out the *$targets* array and set *$targets[0]* to be the same name as the source object.

```
//check for object
        $isObject = `objExists $myTarget`;
```

Next, we do our standard sanity checking. Again using the *objExists* command we can determine if the target is really in the scene or not.

```
//Take proper action based on $onError result
if($isObject)
{//object exists, apply the attributes
  jcApplyKeys($myAnimInfo, $targets, $indexList, $i, ...
    $mySource, $currentIndex, $offsetType, $myOffset, $doScale);
}//object exists, apply the attributes
```

If the target does indeed exist, then we can use the *jcApplyKeys* procedure to actually apply the animation info to the target. We will go over the *jcApplyKeys* procedure in much greater detail a little later on.

```
else if($onError == "onErrorContinue")
{//skip this one and move on
        continue;
```

```
        }//skip this one and move on
```

If the object does not exists, then we need to see what the user wants to do about the error. If the user has selected the "onErrorContinue" radio button, then we simply skip the rest of the loop and move on to the next object.

```
else if($onError == "onErrorCancel")
{//stop on error
   error ($mySource + " does not seem to exist, application ...
                                                    stopped\n");
}//stop on error
```

If the user has selected the "onErrorCancel" radio button, then we need to error out of the script and let the user know which target object did not exist.

```
else if($onError == "onErrorAsk")
{//ask to continue or move on

$onErrorDialog = `confirmDialog -title "Error Finding Object"
        -message ($mySource + " was not found.  ...
                                What would you like to do?")
        -button "Continue"
        -button "Cancel"
        -dismissString "Cancel"
        -defaultButton "Continue"`;
```

Otherwise, if the user selected the "onErrorAsk" radio button, then we need to open up a *confirmDialog* and ask them what they would like to do. This works exactly the same as it did for the "POSE" format from earlier in this chapter.

```
        //process response
        if($onErrorDialog == "Continue")
        {//do next object
                continue;
        }//do next object
        else
        {//stop
                error "User Cancelled Operation";
        }//stop
    }//ask to continue or move on
}//autoload file
```

Once the user has dismissed the confirm dialog, we need to deal with that response. If the returned string is "Continue" then we simply use the *continue* command to skip to the next iteration of the loop. Otherwise, we error out and let the user know that the operation has been cancelled.

```
else
{//use custom mapping
        tokenize($myTargets[$i], $targets);
        jcApplyKeys($myAnimInfo, $targets, $indexList, $i, ...
    $mySource, $currentIndex, $offsetType, $myOffset, $doScale);
}//use custom mapping
```

If the user has not selected the automatic mapping option, then we need to do custom mapping. For this, we tokenize the *$myTargets[$i]* element, and store the individual targets into the *$targets* array. We then call the *jcApplyKeys* procedure with all of it's arguments to actually apply the key information to the targets.

```
                }//do each object
                //close progress window
                progressWindow -endProgress;
        }//Animation file
        print "Done...\n";
}//jcApplyAnim
```

Lastly, we close the progress window, print a quick note to the user that we have finished, and close the procedure.

Now on to the last procedure for the "Load" tab, the *jcApplyKeys* procedure.

```
/*************************************************************

jcApplyKeys

Arguments:

string $myAnimInfo[] - this is the main array that holds the
loaded .jcp/.jca pose file

string $targets[] - the array of targets to apply the keys to

string $indexList[] - the list of positions in the myAnimInfo
array of each object name

string $objIndex - the index of the object to work on

string $mySource - name of the source object to get the values
from (in the array)

string $currentIndex - current index of the source object in the
main info array

string $offsetType - how to handle the offset:
"applyToCurrentFrame" or "applyToOffset"

int $myOffset - how many frames to shift the animation before
applying

int $doScale - if 0 we ignore the scale if 1 we take that into
account

Returns:

        none

Notes:

        none
*************************************************************/



global proc jcApplyKeys(string $myAnimInfo[], string $targets[],
string $indexList[], int $objIndex, string $mySource, int
$currentIndex, string $offsetType, int $myOffset, int $doScale)
{//jcApplyKeys
```

For this procedure to work, we need to pass it a fairly hefty amount of information. First, we have the *$myAnimInfo* array, which you should feel pretty comfortable with at this point. This holds the loaded *POSE* or *ANIMATION* file. Next is the *$targets* array. This holds the list of target object names that we are to apply the animation to. After that, we have the *$indexList* array. This holds the index values for the start of each object in the array. Then the *$objIndex*. This is the index of the object we are currently working on. Following that, we have the *$mySource* string, which holds the name of the source object to get the values from in the array. Next, is the *$currentIndex* argument. This argument is the index of the source object inside the main *$myAnimInfo* array. The *$offsetType* variable tells the script how we need to renumber the frames, if at all. The *$myOffset* argument, tells the script how far to shift the frames if the user wishes to use an offset. Lastly, the *$doScale* option, let's the user copy animation information from a different sized skeleton, to the current skeleton.

```
//grab valType
$myValType = $myAnimInfo[1];

//grab rotation order
tokenize($myAnimInfo[$currentIndex+1], $roBuffer);
//The first thing that we are going to do is grab the value type
//and the rotation order for the current object.

//set rotation order for each target
for($q=0; $q < (`size $targets`); $q++)
{//set rotation order on each object
        //set rotation order for this object
        eval("setAttr " + $targets[$q] + ".rotateOrder " +  ...
                                                $roBuffer[1]);
}//set rotation order on each object
```

Now that we have the rotation order, this is as good a time as any to set it for the current object, since this is an object level change, and not at the attribute level. We set it for each of the targets using the *setAttr* command.

```
//step through each attribute
for($j = $currentIndex+2; $j < ((int)$indexList[$objIndex+1]-1);
                                                        $j+=8)
{//do each attribute
```

Now we can start the main loop. This will loop through each attribute for the object and set the proper values for the keys and tangents. The counting for this is going to take a bit of explaining. First we are going to set *$j* (our counter) to be *$currentIndex+2*. We do this because the *$currentIndex* variable is holding the index value for the object name. We need to grab the first attribute, which is two elements further into the array. The first element being the object name, the second is the rotation order, and the third is the name of the first attribute. Next, we loop through the array until we reach *$indexList[objIndex+1]-1*. This is getting the next object name's index, from which we subtract 1 because we do not want to include the next object name, just the last attribute for the current object. We then need to type cast that number to an int, because the value has been stored into the main array as a string. Finally we increment the counter by 8, every time through the loop because that is how many elements are in the array until the next attribute. If you get confused, you should walk through the saved file, as it is pretty much exactly the same as the array.

```
//clear $myFullSource
$myFullSource = $mySource;

//reset firstFrame
$firstFrame= 1;
```

Now we need to clean up the variables a bit, just in case this is not the first time through the loop.

```
//get attribute name and add to source name
tokenize ($myAnimInfo[$j], $tempBuffer);
$myAttrib = $tempBuffer[1];
$myFullSource = ($myFullSource + "." + $tempBuffer[1]);
```

Now we can take the attribute name string from the *$myInfoArray* and store it into the *$tempBuffer*. Then we take the second word, which should be the actual name of the attribute, and store that into *$myAttrib*. Once we have that, we can add the "." and the attribute name to the source name. That will give us the entire attribute "path". So we will end up with something like "pCone1.translateX". We then store that complete "path" into the *$myFullSource* variable.

```
//get KeyVal and number of frames to look at ($j+1)
tokenize($myAnimInfo[$j+1], $tempBuffer);
$numFrames = ((`size $tempBuffer`) - 1);
```

Now that we have the attribute name, we need to go to the next element in the array and grab the long string that holds all of the frame number/value pairs. We are going to tokenize it into the *$tempBuffer* array. Then we are going to take the size of the *$tempBuffer* and store that into the *$numFrames* variable. The actual number of frames is half of the size of the*$tempBuffer*.

```
for($x = 2; $x <= $numFrames; $x+=2)
{//do each frame
```

This is where we loop through all of the frames and use the *$numFrames* variable as the ending condition of the loop. Recall that *$numFrames* was just filled.

```
//is there a key on this frame?
if($tempBuffer[$x] != "*")
{//apply this key
```

The first thing we need to do for each frame is make sure that we have a key recorded in the file for this frame. If so, then we continue with the loop.

```
{//apply this key
        $frameNumber = $tempBuffer[$x-1];
        $keyVal  = $tempBuffer[$x];
if($firstFrame != 1)
{//get previous key value
        $prevKeyVal = $tempBuffer[$x-2];
}//get previous key value
else
{//prevKeyVal
        $prevKeyVal = $tempBuffer[$x];
```

```
        }//prevKeyVal
```

Once we know we have a key, we store the frame number in *$frameNumber* and the value for the key in the *$keyVal* variable. Then we see if this is the first time through the loop, if it is, then we set the *$preKeyVal* to be the same as the current value, otherwise we get the previous key's value, and store that into *$prevKeyVal* instead. These values we will need to do the scale adaptation later on in the script.

```
if($myValType != "BAKED")
{//do tangents
        //inTanTypes
        tokenize ($myAnimInfo[$j+2], $tempBuffer1);
        $myItt = $tempBuffer1[$x];
```

Now we check to see if the current value type is not "BAKED". If it is not, then we store all of the tangent data for the key, as the "RELATIVE" value types have tangent data, where as "BAKED" values do not. To do this, we are going to tokenize the next element in the array, which will be the "In Tangent Type". Once that is stored into *$tempBuffer1*, we then take the "value" part and store it into *$myItt*. We do the exact same thing for the rest of the animation information:

```
//outTanTypes
tokenize ($myAnimInfo[$j+3], $tempBuffer1);
$myOtt = $tempBuffer1[$x];

//inTanWeights
tokenize ($myAnimInfo[$j+4], $tempBuffer1);
$myItw = $tempBuffer1[$x];

//outTanWeights
tokenize ($myAnimInfo[$j+5], $tempBuffer1);
$myOtw = $tempBuffer1[$x];

//inTabAngles
tokenize ($myAnimInfo[$j+6], $tempBuffer1);
$myIta = $tempBuffer1[$x];

//outTanAngles
tokenize ($myAnimInfo[$j+7], $tempBuffer1);
$myOta = $tempBuffer1[$x];

}//do tangents

//Apply keys to each of the targets
for($tcount = 0; $tcount < (`size $targets`); $tcount++)
{//do each target
```

Now that we have stored off all of the needed animation information, we can start looping through the list of targets and applying the animation information to them.

```
//check to see if target is blank
if($targets[$tcount] == "")
{//no targets
        warning ($myFullSource + " doesn't seem to have any ...
                                        targets. Skipping...\n");
        continue;
}//no targets
```

Next, we need to make sure that the current source object actually has some targets. If it does not, then we warn the user and skip to the next object.

```
//reset vectorCounter
$vectorCounter = 0;
```

Now we do a bit of variable clean up to help us out a little later on in the script.

```
if(jcIsRoot($targets[$tcount])==1 && $firstFrame == 1 && $doScale == 1)
{//might have translate, and we are on the first frame
        //now set $firstFrame to 0 so we don't recalculate the ratio
        $firstFrame = 0;
```

Here we check for three things. First we need to know if the current object is a root joint. We also need to know if the current frame is the first frame. Lastly, we need to check to see if the user wishes to compensate for the scale. If all of these are true, then we move on.

```
//now set $firstFrame to 0 so we don't recalculate the ratio
$firstFrame = 0;
```

First we set the *$firstFrame* variable to zero, since the next time through the loop will no longer be the first frame.

```
//search through source to find translateX,Y,Z
for($aCount = $currentIndex+2; $aCount < ((int)$indexList[$objIndex+1]
                                                -1); $aCount+=8)
{//look at each attribute for this object
```

This next section is all about the scale compensation. In order to do that, we need to get the translateX, translateY, and translateZ attribute values from the source. We will then compare that to the location of the root joint of the target scene and measure the difference. This will give us a ratio that we will then apply to each of the translations to the target root joint. This loop will search all of the attribute names of the current source object looking for the translation attributes. We need all three or we cannot do the calculation.

```
tokenize($myAnimInfo[$aCount], $tempBuffer2);
$thisAttribute = $tempBuffer2[1];
```

Here we tokenize the attribute name line, and store the actual name into the *$thisAttribute* string.

```
if($thisAttribute == "translateX")
{//translateX
        tokenize($myAnimInfo[$aCount+1], $tempBuffer3);
        $sourcePosList[0] = $tempBuffer3[2];
        $vectorCounter++;
}//translateX
```

Once we have the attribute name, we can check to see if it is one of the translation attributes. We are going to check the *$thisAttribute* variable a maximum of three times, once for each translation direction. First up is translateX. If we do have a "translateX", then we tokenize the next element in the array which actually holds the key value for that attribute. We are going to grab the third word in the element, since that will be the value for the first frame, which is the only frame that we are using to determine the scale ratio between the two skeletons.

```
    else if($thisAttribute == "translateY")
    {//translateY
        tokenize($myAnimInfo[$aCount+1], $tempBuffer3);
        $sourcePosList[1] = $tempBuffer3[2];
        $vectorCounter++;
    }//translateY

    else if($thisAttribute == "translateZ")
    {//translateZ

        tokenize($myAnimInfo[$aCount+1], $tempBuffer3);
        $sourcePosList[2] = $tempBuffer3[2];
        $vectorCounter++;

    }//translateZ
}//look at each attribute for this object
```

Here we are checking to see if the attribute is translateY or translateZ. We are looking for all three. If it is translateX, then we put it into *$sourcePosList[0]*, if its Y, then we put it in *$sourcePosList[1]*, and if its Z, then we put it in *$sourcePosList[2]*. We also need to increment the *$vectorCounter* each time we get a match. This way we can determine if we have found all three attributes.

```
if($vectorCounter == 3)
{//get ratio

        $targetPosList[0] = getAbsLoc($mySource, "translateX");

        $targetPosList[1] = getAbsLoc($mySource, "translateY");

        $targetPosList[2] = getAbsLoc($mySource, "translateZ");
```

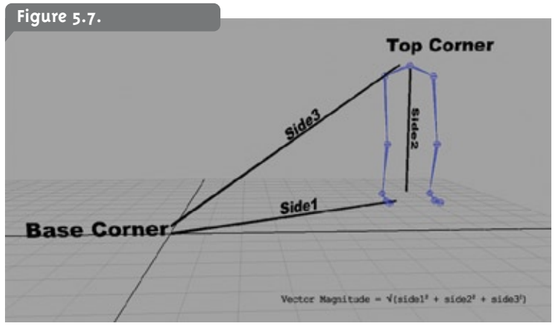
Once *$vectorCounter* is 3, we can now calculate the scale ratio between the root joint of the source and the target skeletons. First we grab the world space coordinates using *getAbsLoc* and store them into *$targetPosList*. Just like for the *$sourcePosList,* we are going to store X into [0], Y into [1], and Z into [2].

```
//calculate the source vector magnitude
$sourceMag = (sqrt((`pow $sourcePosList[0] 2`) + ...
        (`pow $sourcePosList[1] 2`) + (`pow $sourcePosList[2] 2`)));

$targetMag = (sqrt((`pow $targetPosList[0] 2`) + ...
        (`pow $targetPosList[1] 2`) + (`pow $targetPosList[2] 2`)));
```

In order to find the difference in scale, we are going to calculate the magnitude of the vector from 0,0,0 to the root joint of the source object. Then we will find the magnitude of the vector from 0,0,0 to the target object. The ratio between the two will give us a percentage that we can multiply the translation values of the target by, to scale them up or down depending on if the skeleton is bigger or smaller than the source skeleton.

A vector is a collection of three values, which are the lengths of the sides of a right triangle. The base corner will always be a 0,0,0. The top corner will always be at the center of the current skeleton's root joint. The remaining corner will be the "run" from the base corner to the top corner along the ground plane. FIGURE 5.7.

Figure 5.7.



Once we have those lengths, we can calculate the vector magnitude by taking the square root of the first side squared plus the second side squared plus the third size squared.

```
//check for divide by zero

if($sourceMag != 0)
{//get ratio
        //divide to get the percent difference
        $myRatio = ($targetMag/$sourceMag);
}//get ratio
else
{//no ratio
        $myRatio = 1;
        warning("Source Magnitude is 0, using a ratio of 1:1\n");
}//no ratio
```

Now that we have our magnitudes, we need to find the percent difference between them. To do that, we divide the target magnitude by the source magnitude. However, there is a chance that the source magnitude could be zero, if the source root joint is sitting a 0,0,0. If this is the case, we cannot divide by zero, so we set the ratio to 1 and warn the user. Once we have the ratio stored in *$myRatio*, we are done with the scaling part.

```
if($myValType == "BAKED")
{//do abs
        if($myAttrib == "translateX" ||
        $myAttrib == "translateY" ||
        $myAttrib == "translateZ")
        {//relative translate
```

Now we can check to see if our value type is "BAKED". If it is, then we need to check to see if the current attribute is a translation attribute. If this is the case, then we need to check to see if we need to do anything with the scale.

```
        if($offsetType == "applyToOffset")
        {//use offset
                $frameNumber = $frameNumber + $myOffset;
        }//use offset
```

To start with, we check to see if we need to apply an offset to the frame numbers. If *$offsetType* is set to "applyToOffset", then we simply add the new offset to the frame number that is in the saved file.

No we can store that back into *$frameNumber*.

```
//update current frame
currentTime $frameNumber;
```

Now that we have our correct frame number offset, we can update the time slider to the new frame number with the *currentTime* command. This will move the "play head" of the timeline to the specified frame number. In this case, it is $frameNumber.

```
if($doScale)
{//adapt for scale
    $temp = $keyVal - $prevKeyVal;
    $temp1 = $temp * $myRatio;

    $keyVal = (getAbsLoc($targets[$tcount],$myAttrib) + $temp1);
}//adapt for scale
```

Now we can deal with the scale if we need to. If *$doScale* is set to 1, then we apply the scale difference. First, we take the current key value, and subtract from it the previous key value. This will get us the distance moved from the previous frame to the next frame. We then take that and multiply it by our ratio, which will get us a percentage of the distance we need to move. Once we have that new value, we use the *getAbsloc* procedure to get the current position of the joint, then add to it the new distance that we need to move. We then store that into *$keyVal*.

```
//set the new value
setAbsLoc($targets[$tcount], $myAttrib, $keyVal);

//key the new value
setKeyframe -attribute $myAttrib $targets[$tcount];
}//good to go
```

Now we can set the new position with the *setAbsLoc* procedure. Once we have moved the joint to the new location, we can use the *setKeyframe* command to set a key. For this command, all we need to pass to it is the name of the attribute, with the *-attribute* flag, and the name of the object to set the key on.

```
else
{//do relative

        if($offsetType == "applyToOffset")
        {//use offset
                $frameNumber = $frameNumber + $myOffset;
        }//use offset

        setKeyframe -attribute $myAttrib
                -value $keyVal
                -time $frameNumber
                $targets[$tcount];
}//do abs
```

Otherwise, if the current attribute is not a translation attribute, then we simply update the offset if we need to and then use the *setKeyframe* command to set a new key for the object. This time, we will pass-in the attribute name, the value of the key we want to set, the frame number we want to set the value on, and finally the name of the object for which we want to set the key. Since we are not dealing with translation, we do not need to do anything with the scale, or use world space values.

```
else
{//do relative
        if($offsetType == "applyToOffset")
        {//use offset
                frameNumber = $frameNumber + $myOffset;
        }//use offset
        setKeyframe -attribute $myAttrib
                -value $keyVal
                -time $frameNumber
                $targets[$tcount];
```

Now, if the value type we checked for a while back was not "BAKED", then we need to deal with the relative values. The first part is very much the same. We determine our offset and set a key. However since we are dealing with real keyframes, we need to use that tangent information that we read in a while back. So we do the following:

```
                        //adjust the tangent info
                        keyTangent -edit
                                -time $frameNumber
                                -attribute $myAttrib
                                -absolute
                                -inTangentType $myItt
                                -outTangentType $myOtt
                                -inAngle $myIta
                                -outAngle $myOta
                                -inWeight $myItw
                                -outWeight $myOtw
                                $targets[$tcount];
                        }//do relative
                }//do each target
            }//apply this key
        }//do each frame
    }//do each attribute
}//jcApplyKeys
```

In order to set the tangent values, we will use the *keyTangent* command. This is where all of that tangent information comes into play. We pass it all in here. First, we tell it what keyframe to apply it to with the *-time* flag. Then we give it the attribute name with the *-attribute* flag. We then tell it that all of the values we are about to give it are absolute values in the graph editor. We then pass-in each of the different bits of the tangent, the inTangentType, outTangentType, inAngle, outAngle, inWeight, and outWeight. Finally, we pass it the object that we want to set the tangent info for.

That's it really. We have now applied the animation to the scene. I don't know about you, but I think that's a pretty nice thing to be able to do. The hardest part is now over. Take a break, sit back, and let your mind cool down a bit. We are going to go over the "Batch" tab next and all of its recursive goodness.

■ Creating Functionality for the "Batch" tab

Let's move on to the final tab in the utility. Here you will really start to see how making all of those functions into their own independent procedure really pays off. There will be very little actual "new" code for this tab. We will mostly be relying on the other procedures we have written thus far. So let's get started.

Let's finish off the first button in the " batch" tab, the "Select Source File" button:

```
button

        -label "Select Source File"
        -command "jcBatchSource(\"batchSourceFileText\");"
        -width 150
        batchSelectSourceFileButton;
```

As you can see, we are going to call the *jcBatchSource* procedure when this button is pressed. This procedure has one argument and that is the name of the textField to store the path and file name to the source file in. Basically, what we want to happen when we press the button is to get a browse window open for the user to choose the source .jca or .jcp file from. So let's take a look at the procedure itself, starting as always with the comment block:

```
/*************************************************************

jcBatchSource

Arguments:
        string $myField - name of the field to store the selected
        source file

Returns:
        none

Notes:
        This lets the user select the source file for the
        batching process

*************************************************************/
```

As you can see, this only takes one argument and does not return anything when it is done. Also, as stated in the notes section, this simply let's the user select a source file to batch. The procedure declaration looks like this:

```
global proc jcBatchSource(string $myField)
{//jcBatchSource
```

Not too much going on here as you can see. We simply setup one argument called *$myField* which we will use within the procedure itself.

```
        //open a file dialog
        $myPath = `fileDialog -directoryMask "*.jcp; *.jca"`;

        //save path to text field
        textField -edit -text $myPath $myField;
}//jcBatchSource
```

This procedure is actually very simple. First, we open the browse window by calling the *fileDialog* command and giving it a *-directoryMask* of "*.jcp" and "*.jca". This will then only show the user the .jcp and .jca files in a folder. The *fileDialog* command will return the full path to the file. This is then stored into the *$myPath* variable. Lastly, we use the *textField* command in *-edit* mode to change the text in the field to be the contents of $myPath. This should now contain the full path to the file. That is it really, not much to it. This will now let you browse for the source file.

Next up is the "Select Log File" button. This is nearly exactly the same as the "Select Source File" button.

```
button
        -label "Select Log File"
        -command "jcLogSource(\"batchLogFileText\");"
        -width 150
        batchSelectLogFileButton;
```

This time, we are going to call the *jcLogSource* procedure and pass in the name of the textField that we want the path to the log file stored. The procedure's comment block is as follows:

```
/*************************************************************
jcLogSource

Arguments:
        string $myField - the text field that will hold the path
         to the log file

Returns:
        none

Notes:
        This will let the user set a log file path
*************************************************************/
```

Much like the "Select Source File" button, this procedure takes one argument, which is the name of the text field to store the path in, and returns nothing. The procedure declaration is also very similar:

```
global proc jcLogSource(string $myField)
{//jcLogSource
```

The code for this is a little bit different. It is actually more similar to the "Save To File" button from the "Save" tab. Since Maya does not really have a "save file" dialog that we can access, we need to build our own dialog and deal with the response ourselves.

```
//Open a save file dialog

$savePromptDialog = `promptDialog -title "Save File"
                        -message "Specify Path to Save file"
                        -button "OK"
                        -button "Cancel"
                        -dismissString "Cancel"
                        -defaultButton "OK"`;
```

Firstly, we use the *promptDialog* command with the proper flags to open up a dialog window. Then we need to deal with the response:

```
if($savePromptDialog == "OK")
{//get log file path
        $myPath = `promptDialog -q`;

  //save path to text field
  textField -edit -text $myPath $myField;
```

```
                }//get log file path
}//jcLogSource
```

If the value returned from the *promptDialog* command is "OK", then we store the path into the *$myPath* variable by putting the dialog into *-query* mode. Then we use the textField command in *-edit* mode to fill in the textField with the full path to the log file. Now we simply close the procedure. This will now allow the user to specify a filename for the log file.

Next up, we have the "Add Item To List" button. This one is a little bit more complicated because of the different types of items that can be added to the list, along with the recursion option which throws a bit of a wrench into our plans.

```
button
        -label "Add Item To List"
        -command "$myBatchArray = jcAddBatchListItem
             ($myBatchArray,\
             `radioCollection -query -select batchListTypeCol`,\
              `checkBox -query -value batchRecursiveCheckBox`);"

        batchAddFileToListButton;
```

This button calls upon the *jcAddBatchListItem* procedure. This takes an array that holds all of the current "target" files, the current selection of the *batchListTypeCol* radiobutton collection and the current state of the *batchRecursiveCheckBox*. Lets have a look shall we?

```
/*****************************************************************

jcAddBatchListItem

Arguments:
string $batchArray[] - the array that holds the paths to the
items that areto be batched

string $listType - "batchListFolders" if the batchList
textScrollList holds folder names  "batchListFiles" if the
batchList textScrollList holds file names

string $recursive - "true"/"false" value from the recursive
checkbox in the batch tab

Returns:

        string[] - returns the updated batchArray that holds
         the newly added item

Notes:
        Adds an item to the batchList
*****************************************************************/

global proc string[] jcAddBatchListItem(string $batchArray[],
string $listType, string $recursive)
```

This procedure is designed to add an item to the "Batch List". I can hear you saying to your self right now, "Didn't we already write a procedure that adds items to a textScrollList?". To do that, I would have to answer "yes", however there is a bit more to adding an item to this particular list, as you will see. Let's take a look at the arguments. For this procedure, there are three arguments. First is the *$batchArray[]*. This is an array of strings that holds all of the items to be batched so that when we run the "go" button for this tab, we can simply run down the list and do whatever we need to do to the files. Next up is the *$listType* which corresponds to the selected radio button in the "batchListTypeCol" radio collection. This basically is saying weather or not the items in the list are files or folders. Lastly is the *$recursive* string. This is either "true" or "false". If the "List Type" is set to "batchListFolders" then the user will have an option to set those folders to be recursive or not. This basically means that the script will search all of the folders within the stated folder for .ma or .mb files to work with. *Recursion* is something that we will talk about in more depth later, however just so you have some idea of what is going on when we talk about recursion, here is a quick definition:

Recursion is basically the ability of a procedure to call itself. That's right, that means that the function can call upon itself, within itself. Confused yet? Granted, the function needs to be carefully set up so that we do not get into an "infinite loop" situation, but it is possible and can be very useful for going into folders within folders.

Now let's step though each of the parts of this procedure:

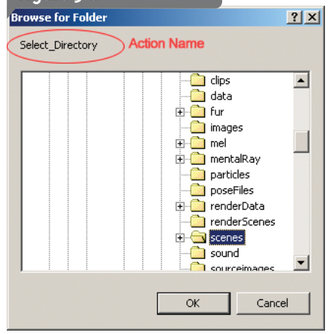
```
if($listType == "batchListFiles")
{//add individual files

        //pop up a file open dialog
        $myPath = `fileDialog -directoryMask "*.mb; *.ma"`;

        //get file name from path
        $numTokens = tokenize($myPath, "\\\/", $tempArray);

        //set variable
        $myFileName[0] = $tempArray[$numTokens-1];

}//add individual files
```

First, we check to see what *$listType* is set to. If it is set to "batchListFile", then we know that our list will only contain files and no folders. So we use the *fileDialog* command again and open a browse window that is looking for "*.mb" and "*.ma" files. Once the user presses the "open" button in the dialog, the path will get stored into the *$myPath* variable. Now that we have the path to the file, we can store it in the list. However we probably do not want to store the entire path in the list. So to store only the file name, we need to pull it out of the path. We are going to use the *tokenize* command again to pull the file name out. The first argument sent to the tokenize command is the *$myPath* variable which is the path that we want to break up. The next argument is the pattern to break up the string with. This one is a little different from the rest that we have used thus far. What we want to do is break up the path on the "\" characters or the "/" characters, assuming a path looks like this: "c:\my\stuff\example\myscene.mb" . All we want is the myscene.mb part, so we will break it up on the "\" characters. However, the slashes are "special" characters. For instance, the back-slash is the "escape" character. So in order to force Maya to see that character as a back-slash, we need to escape the escape character. This is why we have the "\\". The same goes for the forward-slash. We need to escape it by putting a back-slash in front of it. So we end up with "/". So all together we have "\\\/". The last argument is the array that we are going to store all of the pieces in, and in this case, we are going to put them into *$tempArray*. Lastly, we want to take that array and grab the last element in it, which is the name of the file. We access this by taking the number of tokens returned from the *tokenize* command and subtracting one (becuase arrays are 0-based). We take that string and store it into *$myFileName[0]*. We could store this into a single string instead of storing it into an array, but later on down, in the procedure, we will be using the *jcAddToList* command to actually add the name to the textScrollList, and that procedure takes an array of items to add.

```
else if($listType == "batchListFolders")
{//add folders

   $doCallback = "jcAddPathCallback";
   fileBrowserDialog -m 4 -fc $doCallback -an "Select_Directory";
```

That's basically it for files, however, folders are little more tricky. If the *$listType* is set to be "batchListFolders", we execute the above part of the script. In order to browse for folders, we need to do things a little differently than when we browse for files. For this, we are going to use the *fileBrowserDialog* command in Maya. Firstly, we use the *-mode* flag to set the mode to 4, which means that we want to browse for folders only. Then we need to use the *-fileCommand* flag. This flag tells the command what procedure to call when the dialog box is closed. The *fileBrowserDialog* command does not actually return anything useful to us, but it will call a procedure (called a callBack procedure) and automatically pass to that procedure two arguments, the filename and the file type. We are also using the *-actionName* flag. This will allow us to write a quick message out to the user near the top of the browse dialog.. FIGURE 5.8.

Figure 5.8.



Once the "OK" button is pressed in the dialog, the *$jcAddPathCallback* procedure will be called. Let's take a quick look at that:

```
/*****************************************************************

addPathCallback

Notes:
        This is called when automatically from the
        fileBrowserDialog proc built into Maya
        It essentially stores the selected folder
*****************************************************************/

global proc jcAddPathCallback(string $filename, string $fileType)
{//jcAddPathCallback
//global vars
        global string $mySelectedFolder;
        //record the selected folder
        $mySelectedFolder = $filename;
}//jcAddPathCallback
```

As you can see in the procedure declaration, it is looking for two arguments, which is good, because the *fileBrowserDialog* command is going to automatically pass in two arguments to the procedure. In this case, we are really only ever going to use one of them, the *$filename*. Firstly, we are going to declare a global variable called *$mySelectedFolder*. Any procedure within Maya will now be able to see that variable. This variable is going to store the contents of *$filename* . That is it really. We needed to do all of that to simply get the path to the folder selected in the *fileBrowserDialog* command. Now let's get back to the rest of the of the *jcAddBatchListItem* procedure:

```
   //get folder name from path
   $numTokens = tokenize($mySelectedFolder, "\\\/", $tempArray);

   //set path variable
   $myPath = $mySelectedFolder;

   //set variable
   $myFileName[0] = $tempArray[$numTokens-1];


}//add folders
```

Here we are going to do the same thing as we did with the file name earlier. We are going to use the *tokenize* command to separate out the folder name from the whole path. We are then going to store that into the *$myFileName* array as well at the [0] index.

```
else
{//bad type
        error ("jcAddBatchList can't handle the type:" ...
                                                + $listType + "\n");
}//bad type
```

Now we need to add-in that little bit of error checking, and if the *$listType* is somehow not set to the two different strings we are looking for, then the script errors out and tells the user.

Now as with the other places where we add items to a textScrollList, we need to check for duplicates:

```
//check for dupes in array
for ($each in $batchArray)
{//check for dupes
        if($each == $myPath)
        {//add to list
                //set duplicate flag
                $isDuplicate = 1;
        }//add to list
}//check for dupes
```

Here we are going to use a *for-in* loop and check each item in the current *$batchArray* with the path that the user just selected. If that path is already in the array, then the *$isDuplicate* variable gets a 1 stored in it. Otherwise, *$isDuplicate* stays at 0. Next we actually add the path to the array and the textScrollList:

```
if($isDuplicate != 1)
{//not a dup
        //add whole path to batchFileArray
        $batchArray[`size $batchArray`] = $myPath;

        //add filename to path
        if($recursive)
        {//mark as recursive
        $myFileName[0] = ($myFileName[0] + " (*)");
        }//mark as recursive

        jcAddToList($myFileName,"batchFileList", 1);

}//not a dup
```

If the *$isDuplicate* variable was never set to 1, then we can add the path to the array and the list. We do this by storing the path into the array at one index past the last index. Maya will resize the array to accommodate the new item. So we do this:

```
$batchArray[`size $batchArray`] = $myPath;
```

First, we get the size of the *$batchArray* and then use that as the index number. The size of the array will always be one index past the last index in the array. Then we simply store *$myPath* into that index, which will add our new path to the end of the *$batchArray* array. Now we need to add the new item to the end of the textScrollList as well. To do that, we first need to check to see if *$recursive* is set. If the user has checked the "Recursive" box in the UI, then we need to mark the newly added folder as a recursive folder. We are going to do this by adding an "(*)" to the end of the folder name. This will delineate the folder as being a recursive folder, meaning that we are to search that folder for other folders within it. Lastly, we actually add the new item to the list by using the *jcAddToList* procedure like before. The only difference this time is since we have checked for duplicates already, we need to pass in a 1 for the "ignore duplicate" argument of the *jcAddToList* procedure. If we did not set it to ignore duplicates, it would end up removing to files that might have the same filename, but live in different folders.

```
{//its a dup
        warning ($myPath + "was not added, because it seems ...
                                to be in the list already. \n");

}//its a dup
        //return the array
        return $batchArray;
}//jcAddBatchListItem
```

Now, if *$isDuplicate* was indeed set to 1, then we simply warn the user that the path is already in the list, and we skip the rest of the code to add the item to the list. Finally, we return the newly edited *$batchArray* to the caller and close the procedure.

Since we have a procedure to add and an item to the "batch list", then we are going to need a procedure to remove an item as well:

```
/*************************************************************

jcRemoveBatchListItem

Arguments:
string $batchArray[] - the array that holds all of the items
                        that are to be batched

string $myList - the textScrollList that is the batchList

Returns:
        string[] - returns the updated batch array

Notes:
        This will remove an item from the batchList
*************************************************************/


global proc string[] jcRemoveBatchListItem(string $batchArray[],
string $myList)
{//jcRemoveBatchListItem
```

First up, the arguments. This procedure is going to take two arguments. The first one is the *$batchArray* that is currently holding all of the paths to all of the files in the "batch list". The second argument is a string called *$myList*. This is simply the name of the textScrollList that we are using as the "batch list". This procedure will return the updated *$batchArray* after the items have been removed.

The main concept for this procedure works like this. Since Maya does not provide us with a good way to remove items out of the middle of an array, we are going to need to make a copy of the array. Then "mark" certain items in that array to be deleted. Then we clear the original array, and copy back all of the items that have not been marked for deletion. Hehe, "Marked for DELETION!"... sounds bad doesn't it ?

```
//We make a copy of the array...
//make copy of array
        $tempArray = $batchArray;
```

Now we need to get a list of all of the items the user has selected to be removed from the list. Here we use the *textScrollList* command in *-query* mode and use the *-selectIndexedItem* flag. This will return a list of the "slots" in the textScrollList that are selected. Please note that the first slot is 1 not 0.

```
//get selection list
$mySelection = `textScrollList -query ...
                                -selectIndexedItem $myList`;
```

Here we actually "mark" the items that we want to delete. To do this, we are simply going to replace the item in the array with the string "*DEL*". Then when we go to copy the good parts back we will skip any element that contains "*DEL*". We use the list of "slots" stored in *$mySelection* minus 1 (because the items in *$mySelection* start at 1 not 0) as the index of the item to be removed in the *$tempArray*.

```
for($i = 0; $i < `size $mySelection`; $i++)
{//remove from list
        //remove from array
        $tempArray[$mySelection[$i]-1] = "*DEL*";
}//remove from list

//clear the old array

clear $batchArray;
```



```
//Copy back only items that are not "*DEL*"

//copy new list of files back
for($i = 0; $i < `size $tempArray`; $i++)
{//copy array back

        if($tempArray[$i] != "*DEL*")
        {//do copy
                $batchArray[`size $batchArray`] = $tempArray[$i];
        }//do copy
}//copy array back
```

To accomplish this, we are going to use a for loop to step through each element of the *$tempArray*. Then, if the *$tempArray[$i]* is not equal to "*DEL*", we need to copy that element to the end of the *$batchArray*. Just like before, we can find the end of an array by taking its size and using that as the index to store the new value into. Once we have finished with the loop, all of the "good" items should have been copied back. Now all that is left is a bit of clean up:

```
//remove from list
jcRemoveFromList($myList);

//return the new list of files
return $batchArray;
}//jcRemoveBatchListItem
```

First, we remove the item from the textScrollList using *jcRemoveFromList*. Then we simply return the newly created *$batchArray* and close the procedure.

Now we move on to the final part of the "Batch" tab. This is the "Start Batch" button. Let's take a look at the *-command* flag for this button before we jump right into the heart of what this button does.

```
button

        -label "Start Batch"

        -command "jcStartBatch($myBatchArray,\
        $myFileArray,\
        `radioCollection -query -select batchListTypeCol`,\
        \"batchFileList\",\
        `textField -query -text batchSourceFileText`,\
        `textField -query -text batchLogFileText`,\
`radioCollection -query -select batchLoggingStyleRadioCol`,\
`radioCollection -query -select batchLoggingPrefRadioCol`);"


        batchStartBatchButton;
```

This button basically just calls the *jcStartBatch* procedure and passes in a bunch of arguments. The first one is our batch array that holds all of the items to be batched. Next is the *$myFileArray* which is the same as the one in the "Load" tab. We are going to be using some of the procedures from the "Load" tab to load-in the source file and apply it to the target files, so we will need to give the procedure access to the *$myFileArray*. Next is the selected option for the "List Type". Following that is the name of the textScrollList to get the list of items from. In this case, it is the "batchFileList". Notice the escape characters before each of the two double quotes. Next is the name of the source file, which we will get from the first text field at the top of the tab. Then we pass-in the log file, which we get from the textField just below the "Source File" textField. Then we pass in the selected option for the "Logging Style" and finally the selected option for the "Log File Preference". Let's take a look at how it all works:

```
/*************************************************************
jcStartBatch

Arguments:
string $batchArray[] - the array that holds the items to be
batched

string $myAnimInfo[] - the array that holds the animation/pose
info to be used as the source file

string $listType - "batchListFiles" or "batchListFolders", what
kind of items are in the batchList

string $batchList - the name of the textScrollList to use as the
batchList

string $sourceFile - path to the source file to be applied to the
batched items

string $logFile - path to the log file

string $logStyle - "batchErrorsOnly" or "batchVerbose", log only
errors, or log everything

string $logPref - "batchOverWriteLog" or "batchAppendLog",
overwrite the old log or add to it

Returns:
        none

Notes:
        This will start the batching process based on the info
in the batch tab
*************************************************************/
global proc jcStartBatch(string $batchArray[], string
$myAnimInfo[], string $listType, string $batchList, string
$sourceFile, string $logFile, string $logStyle, string $logPref)
{//jcStartBatch
```

Not too much going on here other than the ton of arguments being passed in. Let's jump in to some of the real code:

```
//make sure there are some files listed
$allListItems = `textScrollList -query -allItems $batchList`;
if(`size $allListItems` == 0)
{//error, no files
        error ("No Files listed in the Batch List!\n");
}//error, no files
```

The first thing we want to do is make sure that there are items in the textScrollList. We use the *textScrollList* command with the *-allItems* flag to return the names of all of the items and store them into the *$allListItems* array. Then we check the size of the array and make sure that it is not equal to 0. If it is, then we error out and tell the user that there

are no files in the list.

```
//which type are we dealing with?
if($listType == "batchListFiles")
{//batch the files
        $allFiles = $batchArray;
}//batch the files
```

Next we check to see what "List Type" we are dealing with. If the list only contains files, then we simply take the *$batchArray* and copy the whole thing as is into the *$allFiles* list which we will be using later on in this procedure.

```
//folders
else if($listType == "batchListFolders")
{//batch the folders
```

If the *$listType* is not "batchListFiles", then we need to check to see if it is the "batchListFolders". If it is, then we need to go through all of the items in the list.

```
//loop through each item in the list

for($i=0; $i<(`size $allListItems`); $i++)
{//do each dir
        $thisItem = $allListItems[$i];

        tokenize($thisItem, "(", $tempBuffer);

        if($tempBuffer[1] == "*)")
        {//do recursion

          //add slash to current path
        $thisDir = ($batchArray[$i] + "/");

          //get files from all folders below this one
        $tempFiles = listMyFiles($thisDir, $allFiles, 1);

        //add to total file list
         appendToArray($allFiles, $tempFiles);

    }//do recursion
```

Since we are now dealing with a list of folders, we need to check each item in the list and see if it has been marked for recursion. So we take the name of the item from *$allListItems[$i]* and store it into $thisItem. Once we have the name, we are going to tokenize it and break it up based on the left parenthesis "(". Once we find that, we need to check the second "token" and see if it is "*)". If it is, then we can be sure that this item has been marked for recursion. Now we need to get the full path to that folder from the *$batchArray*. Each full path is at the same index as the name of the folder in the textScrollList. We simply need to check the [$i] index and that will be the path the folder listed in the textScrollList. Once we have that, we need to add a final "/" character to the end of the path because we are dealing with folders and not filenames. With the full path in hand, we can finally grab all of the files from that path. We are going to do this with the listMyFiles procedure which we will get into later on. For now, all you need to know is that it takes three arguments. The first being the current full path to the folder. The second argument is the $allFiles array which holds the full paths and file names to all of the files that are going to be batched.

Lastly, is the option to do recursion on the folder or not. If this is 1,then recursion is turned on. If it is set to 0, then recursion will not be used, and only the files in the specified path *($thisDir)* will be batched. This procedure is going to return the full path and file names to all of the .ma and .mb files that it found in the specified folder. This is where we take that array that it returns and we add it to the end of the *$allFiles* array which, again, houses all of paths and file names that are to be batched. We then append the items to the end of the array by using the *appendToArray* procedure. We will also get to this at the end of this section. This procedure takes two array, and tacks the second one on to the end of the first one.

```
else
 {//do each dir
  //add slash to current path
  $thisDir = ($batchArray[$i] + "/");

  //add files to total file list
  appendToArray($allFiles, listMyFiles($thisDir, $allFiles, 0));
 }//do each dir

}//do each dir
}//batch the folders
```

If the folder does not need to be done recursively, then we simply tack on a "/" to the end of the path and then use the *listMyFiles* procedure with the recursive option set to 0. This will return an array of paths and file names which will then be used by the *appendToArray* procedure to stick that list onto the end of the *$allFiles* array.

```
else
{//unknown type
        error ("jcStartBatch can't handle the type: " ...
                                                + $listType + "\n");
}//unknown type
```

As usual, we add a touch of error checking just incase this procedure is used in other scripts. If the *$listType* does not match the first two, then we error out of the script and tell the user.

Now that we have our list of files that we want to operate on, we need to setup all of the logging options before we start the batch run:

```
//Setup the batch run
if($sourceFile != "")
{//we have a source file
```

First, we make sure that there is a source file filled in. If there is not, then the field will be blank. If this is the case, then we error out of the script and tell the user.

```
if($logFile != "")
{//start logging
```

Next, we check to see if the user has filled-in a log file. If so, then we start the logging, otherwise we skip all of the logging stuff, and print "No Logging" out to the script editor. If the user has filled in the log file name, then we setup all of the logging stuff:

```
//setup echo to file
if($logPref == "batchAppendLog")
{//append
        //do nothing, as this is the default for the
        //scriptEditorInfo function
}//append

else if ($logPref == "batchOverWriteLog")
{//overwrite

        //clear out old log
        $logId = `fopen $logFile "w"`;
        fprint $logId ("Animation Copy & Store Log File\n");
        fclose $logId;
}//overwrite
```

The first part of the logging setup is determining if the user wishes to overwrite the log file (if it already exists) or simply append to the previous one, again assuming it already exists. We can determine what the user has selected by checking the contents of *$logFilePref* which is an argument that is passed into this procedure. If the contents of *$logPref* is "batchAppenLog", then we actually do nothing. Otherwise, if it is equal to "batchOverWriteLog", then we need to clear the old file first. To do that, we simply use *fopen* to open the logfile for write, then *fprint* to a single line of text. This will essentially clear out the file. Then we use *fclose* to actually save the file to disk.

Next we check to see what logging style the user wants to use:

```
if($logStyle == "batchErrorsOnly")
{//echo errors only
        scriptEditorInfo -historyFilename $logFile ...
                                -writeHistory true -suppressInfo;
}//echo errors only
else if($logStyle == "batchVerbose")
{//echo all info
        scriptEditorInfo -historyFilename $logFile ...
                                                -writeHistory true;
}//echo all info
```

We can check this by again using one of the arguments that are passed to this procedure. This time, we check *$logStyle* to see if it is equal to "batchErrorsOnly". If it is, then we need to use the *scriptEditorInfo* command to set up the logging. What we are really doing is setting the scriptEditor to mirror everything that is printed to it, to a file. Here we give it four flags. The *-historyFileName* flag is the name of the file that the information printed to the script editor will be mirrored to. The next flag is the *-writeHistory* flag. This tells Maya to write the information to the file stated in the *-historyFilename* flag. Next is the *-suppressInfo* flag. This tells Maya not to write out info that is not an error or a warning. This will effectively write-out only errors.

If the *$logStyle* variable is set to "batchVerbose", then we are not going to suppress anything. Everything will be written to the file. The code for that is exactly the same, minus the two suppression flags.

Now that we are writing the batch info to a log file, we are going to want to give the user a bit more information:

```
$systemTime = `system("time /t")`;
$systemDate = `system("date /t")`;
print ("Batch Process Started on " + $systemDate);
print ($systemTime + "\n");
```

We are going to use the *system* command to got get the time and the date that the batch process was started. The system command will allow Maya to send commands directly to the Windows shell. We are sending the "time /t" and the "date /t" command. These will return thedate and the time from the Windows command shell. Next, we simply print out the date and time.

```
//start timer
        $startTime = `timerX`;
}//start logging
```

Lastly, we start the timer for the batch process. Using the *timerX* command, we can measure how long something takes in Maya. This "timer" has two parts. The first is the start of the timer. This time is stored in memory if the *timerX* command has no flags. Later, when we want to get the elapsed time, we use the timerX *-startTime $startTime* command. We will see this a little later on.

Now that the logging has been taken care of, we can actually start the batch process. First we need to read the source file into memory:

```
//Load the current file
$myAnimInfo = jcLoadAnimFile("", $sourceFile)
```

For this, we are going to use the *jcLoadAnimFile* procedure that we wrote for the "Load" tab. Aren't you glad we made this a procedure so we did not have to figure out how to do all of that again? We are going to store the resulting array into the *$myAnimInfo* array for later use.

```
//grab the file type
$myFileType = jcGetFileType ($myAnimInfo);

//grab the value type
$myValueType = jcGetValueType ($myAnimInfo);

print "Start batching!\n";
```

Next, we grab the file type with the *jcGetFileType* procedure, and we grab the value type with the *jcGetValueType* procedure. Then we simply tell the user that we are starting the batch process.

```
//Run the batch here:
//do this to all files in total file list
$batchTotal = (`size $allFiles`)
for($each in $allFiles)
{//do each file
        print ("\nFile: " + $each + "("+ $batchCount ...
                                + " of " + $batchTotal + ")\n\n");
```

Here we are setting-up the main loop that will walk through each of the files in the list to be batched and apply the animation or pose. We are also giving the user a bit more information about what is going on. We are printing out the current file name and what file number we are on out of the total number of files to be processed. Now we are ready to open the first scene to be batched:

```
//open target file
if(catch($thisFile = `file -force -open $each`))
        {//oops
                warning ("Was unable to open: " + $each + " for
```

```
batch process...\n");

//update the count and continue
$batchCount++;
continue;
}//oops
else
{//file opened
        print ("Opened " + $each + " for batch process\n");
}//file opened
```

We will open the file with the *file* command. We pass it the *-force* flag as well as the *-open* flag. The *-force* flag will allow Maya to open the file without asking for any intervention from the user. The *-open* flag tells the file command to open the file into Maya. We are also using a new command call catch. This command will "catch" an error and allow the script to decide what to do instead of just stopping the script entirely. We are going to use this incase the file *-open* does not work correctly. This will allow us to simply skip that file and move on. However, if the open is successful, then we simply tell the user which file has been opened.

Next, we actually apply the animation or pose to the open file:

```
//apply the animation.. Use all automatic settings
jcApplyAnim($myAnimInfo, $noTargets, 1, "onErrorContinue", ...
                                0, "applyToCurrentFrame", 0);
```

This should look pretty familiar by now. We are simply going to use the *jcApplyAnim* procedure to apply the animation to the open file.

```
//savefile
if(catch(`file -save`))
        {//save didn't go well
                warning ("Couldn't Save " + $each + "\n");
                //update count
                $batchCount++;
                continue;
        }//save didn't go well
        else
        {//all done
                print ("Saved: " + $each + "\n");
        }//all done
        //update batch count
                $batchCount++;
}//do each file
```

Now that we have processed the file, it's time to save the scene back out. Again, we are going to use the *catch* and *file* combination. This time, however, we simply pass the *-save* flag to the *file* command will instruct Maya to save the file. If the save is not successful, then we warn the user and move on. Otherwise, we tell the user the file was saved and move on. We then update the count and go through the loop again.

```
//stop logging
if($logFile != "")
{//calc time
        $elapsedTime = `timerX -startTime $startTime`;
        print ("****Batch Finished! (" + $elapsedTime ...
                                        + " seconds) ****");
                scriptEditorInfo -writeHistory false;
        }//calc time
   }//we have a source file
}//jcStartBatch
```

Once we have gone through each of the files, we then see if logging was turned on. If it was, then we calculate the elapsed time. For this, we again use the *timerX* command and give it the start time. This will cause the command to return the elapsed time from the start time until *timerX* is called again. We are going to store this value into *$elapsedTime*. Next, we tell the user that the batch process has finished and we print-out how long the process took. Then we simply close the procedure. That about does it for the "Batch" tab, however I have waved my hands and said, "Trust me, this procedure will work" a couple of times. So let's blow away some of the smoke, and take down the mirrors to see what is really going on. First up, the *listMyFiles* procedure.

```
/*************************************************************

listMyFiles

Arguments:
        string $startPath - the path to start the file search
        with
        string $filesArray - the array that holds the paths to
        the files
        int $doRecursion - 1 walk through the paths recursively
                                0 only use the top level path to
                                                look for files

Returns:
        string[]  - returns all of the .ma or .mb files
                        found in the search

Notes:
        This returns all of the .ma or .mb files found in a
        given path
*************************************************************/

proc string[] listMyFiles(string $startPath, string
$filesArray[], int $doRecursion)
{//listMyFiles
```

As you can see, this procedure takes three arguments. The first is the path to the folder to start the search in. The second argument is the array that is holding all of the paths and file names that will be batched. Lastly, the *$doRecursion* option. If this is set to 1, then the procedure will search through the folders recursively. Otherwise, it will only get the names of the files in the folder passed into the *$startPath* argument. This procedure will also return all of the .ma and .mb files that it found while searching the given path. So let's take a look at what is really going on .

```
if($doRecursion == 1)
{//do recursion
       //get list of dirs
       //add slash if needed

       $pathLength = size($startPath);
       if(`substring $startPath $pathLength $pathLength` != "/")
{//add slash
        $startPath = ($startPath + "/");

}//add slash
```

We check to see if we need to recurse the folders that we are searching through. Then we check to see if the path ends with a "/" character. If it does not, then we add one to the end by concatenating it onto the end of the $startPath string. We are actually checking to see if the last character is a slash by using the *substring* command. We pass to it the *$startPath* and then the positions of the first and last characters we want returned to us. In this case, we only want the last one so we pass in the *$pathLength* variable for both the first and last character.

```
//make sure its a dir, not a file path

if(`filetest -d $startPath`)
{//its a dir
        $myDirs = `getFileList -folder $startPath`;
```

Next we want to do a bit of sanity checking. We need to check to make sure that the path passed to us is indeed a folder and not a file. So we use the *fileTest* command with the *-d* flag. This float will return true if the path passed to it is a directory, which is what we want.

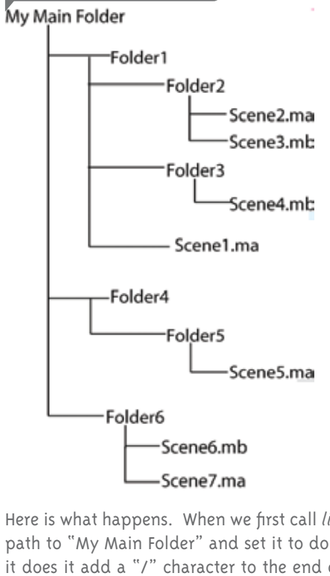
```
$myDirs = `getFileList -folder $startPath`;
```

Now that we know we have a directory, we can use the *getFileList* command with the *-folder* flag to return a list of all of the folders in this particular folder.

```
    //go into each dir
   for($each in $myDirs)
   {//look in each dir
          //build complete path
          $wholePath = ($startPath + $each);

          //start recursion
          $tempArray = listMyFiles($wholePath, $filesArray, 1);

          //combine lists
          appendToArray($filesArray, $tempArray);

   }//look in each dir

  }//its a dir
}//do recursion
```

Once *$myDirs* has been filled, we can start looking for files and other folders in each of the directories in the *$myDirs* array. We are going to use a for-*in* loop to step through the items in the *$myDirs* array. The first thing we need to do is create a complete path with the new folder on the end of it. To do that, we simply take the *$startPath* and tack on the contents of *$each* which is the current item in the *$myDirs* array. The next line might be the most difficult line to get your head around in this *entire utility*, so do not fret if it does not make total sense at first. Just keep looking at it and working through the code on paper. This is what is called recursion. This is when you call a procedure from within itself. In this case, we are calling *listMyFiles* within the *listMyFiles* procedure. This is a difficult thing to explain in English, but here goes. Basically, this is going to keep going deeper and deeper into the folders until it finds a folder that does not contain any other folders. Once this is reached, the *for-in* loop is skipped because there are no items in the *$myDirs* array. We then run the next bit of code which finds all of the files in that folder that does not have any other folders. The execution then returns to the calling procedure which is another instance of *listMyFiles*, which then moves on to the next directory in its own copy of the *$myDirs* array. It once again calls itself if there are other folders within the current folder. This continues until all of the folders in each of the instances of the *listMyFiles* proc have been looked at. Please have a look at the image below. FIGURE 5.9.

Figure 5.9.



Here is what happens. When we first call *listMyFiles*, we will pass it the path to "My Main Folder" and set it to do recursion. So the first thing it does it add a "/" character to the end of the path name. Then we use the *getFileList* command to list all of the folders in the directory and store them into *$myDirs*. So now in *$myDirs* we have "Folder1", "Folder4", and "Folder6". So now we enter the loop that looks into each of the folders for files. The first folder in the list is "Folder1". Next we call *listMyFiles* again (within itself), and the process starts over. As a quick note, it is vitally important to remember that each time we call a procedure, the variables inside that procedure, are local to that procedure only. So when we store items in variables in different "instances" of a procedure, they do not get overwritten, it is just another copy of the same variable. *listMyFiles* has the path to "Folder1" passed to it. So, it tacks on a "/" and uses *getFileList* to get all of the folders within that folder. This time it returns "Folder2" and "Folder3". So we enter the loop that looks into each of the folders and call *listMyFiles* again. First up is the "Folder2". So this time, we tack on a "/" and call *getFileList*. This time it does not return any folders. So when we get to the loop, there are no items in *$myDirs*, so the loop is skipped. Then we look for the files themselves:

```
//make sure it is a dir

if(`filetest -d $startPath`)
{//its a dir

    //get a list of .ma and .mb files for the current dir
    $myFiles = `getFileList -folder $startPath -filespec "*.ma"`;
```

```
$myTempFiles = `getFileList -folder $startPath -filespec "*.mb"`;

//combine files
appendToArray($myFiles, $myTempFiles);

//add full paths to the array (put path in front of the filename)

for ($q=0; $q < `size $myFiles`; $q++)
{//add start path

        $myFiles[$q] = ($startPath + $myFiles[$q]);

}//add start path
}//its a dir
```

Now, since we skipped the loop this time, we can finish the code in the *listMyFiles* procedure. Here we use the *getFileList* command with the *-filespec* flag which forces the command to return the names of the files with the .ma extension. Then we do it again with the .mb extension. Finally, we combine the two lists of .ma files and .mb files into one array, which is the *$myFiles* array. Then we take each of those file names and add the full path to them in the for loop. Once that is done, we return the *$myFiles* array to the caller and then close the procedure.

```
//return the resulting array
return $myFiles;
}//listMyFiles
```

Cool, so now the execution returns to the place where *listMyFiles* was just called from. Thi is of course, a previous copy of *listMyFiles*. In that instance, we move on to the second folder in the list which is "Folder3". We then search it for folders, find out that there are no folders inside "folder3" and then get all of the files in that folder. Then once again, execution returns back to *listMyFiles*. At this point, we have finished the loop for that instance of *listMyFiles*. This means that we now get all of the .ma and .mb files within "Folder1". Then we return all of those files. Now we have finished another instance of *listMyFiles*. Which brings us back to the very first time we call *listMyFiles*. We continue that loop and look into "Folder4", which takes us into "Folder5". We then return "scene5.ma" and back our way back up to the first instance of *listMyFiles*. This proceeds to look into "Folder6" and so on until all of the folders have been looked into. If you do not fully understand this at first, keep working your way through the code and follow along on a sheet of paper with a directory structure on it. Within a few times through, you will understand what is going on. I promise!

One last procedure to go over for this tab. This is the *appendToArray* procedure. This simply sticks, or appends, one array to the back of another.

```
/*************************************************************

appendToArray

Arguments:
        string $array1[] - the first array
        string $array2[] - the array to be tacked on the end of
        $array1

Returns:
        string[] - returns the combined arrays

Notes:
        Appends an array to the end of another array
*************************************************************/

proc string[] appendToArray(string $array1[], string $array2[])
```

This procedure takes two arguments. The first is the array that that second argument will be appended to. This makes the second argument the array that will be tacked onto the back of the first argument.

```
{//appendToArray
        //combine the two sets of files
        $j=(`size $array1`);
        for($i=0; $i<(`size $array2`); $i++, $j++)
                {//append
                        $array1[$j] = $array2[$i];
                }//append

        return $array1;
}//appendToArray
```

This procedure is actually pretty simple. We simply grab the size of *$array1*. Then we loop through *$array2* and stick each element of *$array2* onto the back of the ever growing *$array1*. There is one interesting bit in this code. If you look at the last part of the for loop, you will see that we are updating two counters. Just something to keep in mind as being possible when you are working on your own scripts. We store the size of *$array1* into *$j* and use that as the index number for *$array1*. Then we use *$i* as the index counter for *$array2* which will start at zero. Then we just copy each element over one at a time. That is it really.

Final Thoughts:

Congratulations! We have completed the "Batch" tab. This means we are all done! Whoo hoo! Now I encourage you to start tweaking this utility to add features that you would like. This chapter has covered a huge amount of ground. We have learned everything from making a window, to filling it with a UI, to actually making the UI do something that I think many people will find very useful. Hopefully after working your way through this chapter, as well as this entire book, you should be well on your way to being able to write very useful utilities, all on your own. If nothing else, this should have given you enough information so that you can take information from the web and other sources and really understand what is going on. Now start putting some of those theories into practice within Maya! Remember, all big problems, are simply made up of lots of small problems. Take them one at a time and you will be fine. The more you work with the code and try to make it do what you want, the deeper your understanding will be. Play with the code as much as you can, that's all I ask. Good luck!

Chapter 6
Muscles and Fat

Introduction to Muscle and Fat Deformations in Computer Animation:

This chapter will focus on deformation techniques that will help you take your creatures to the next level. As film and video game audiences begin demanding more believability from their digital thespians, you can rest assured that proper muscle and fat deformations will be at the top of the 'must-have' list. As video game hardware and software capabilities skyrocket over the next generation, technical directors will be borrowing more and more techniques from the bag of tricks currently used in film effects. Most notably, this will include much more realistic deformations in the face and throughout the body.

When most people first start rigging creatures, they are amazed at how quick and easy smooth binding is. Indeed, smooth skinned meshes are easy to work with, fast to solve and perfect for export to game engines. Unfortunately, smooth binding alone, will *never* be able to fully simulate the complex interactions between flesh and bones. This is something that film effects artists have always known. To combat this inherent limitation, they have resorted to layering smooth skinning with other custom deformers to try and simulate muscle bulging and complex muscle-to-muscle collisions.

Up until recently, these 'advanced' techniques have been completely off limits to those in the video game industry. But this is about to change...

There are several factors that inhibit video games from implementing the layered deformation based techniques currently used in films. Firstly, there is the problem of programming a game engine to support more advanced deformers. We are currently seeing a major transition in this area as more and more engines begin to adopt the use of blendshapes in conjunction with smooth binding. The second (and most difficult) problem that video games must overcome is that of current hardware limitations. Layered deformations require significantly more calculations that can result in lower frame rates. Judging by the computing specs stated for upcoming consoles, this will no longer be a problem.

We live in a very exciting time for computer graphics and games. For the first time ever, video games may finally reach that holy grail where it is impossible to distinguish between pre-rendered and realtime graphics. Make no mistake, if we are to reach this level, it will only be because of the hard work of many people like yourself. The concepts presented in this chapter can help you prepare your productions to enter into this new generation

Specifically, we will cover:

- Complete discussion of current techniques.
- Pose Space Deformation (PSD)
- Multi-Axis Drivers
- Using cgTkShapeBuilder.mel to sculpt corrective shapes.
- Detailed coverage of sculpting techniques.
- Using cgTkDynChain.mel to simulate jiggle.
- MEL scripting. The creation of cgTkShapeBuilder.mel

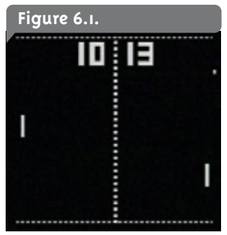
A Brief History of Deformations:

3D In a Nutshell

Our industry is currently in fast-forward. Perhaps it is this ever-changing environment that attracts us to the profession. Whatever it is, you really have to stay on top of things to avoid falling behind. So while most history lessons start hundreds or even thousands of years ago, this one starts in 1958.

During the late 1950's, computer science was the hot new research field. Computer's were being programmed for all sorts of military purposes. They could project bomb trajectories and decipher coded enemy messages. All cool stuff, but the coolest was, by far, (in this humble geek's opinion) the invention of the video game. Specifically, the tennis game known as 'Tennis for Two'. 'Tennis for Two' was later remade into the well known 'Pong' and featured some pretty state of the art particle dynamics (for the time) FIGURE6.1.

Figure 6.1.



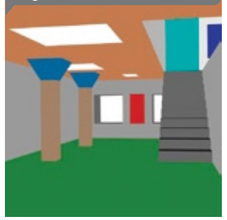
In fact, the white blob of pixels that represented the ball in 'Tennis for Two' might be considered the first video game character ever! Over the next several decades, video game character's remained static pixel images (or *sprites* as they became known as). Each year, video game companies boasted about featuring more colors at increasing resolutions. Games continued evolving in this manner all the way up to the early 1990's. With the release of the brand new 3d capable consoles from Sony and Nintendo, the mid-nineties were a frantic race.

3d animation can be traced far back before the mid 1990's, but it was during this time that the technology matured into an formidable market force. In 1995, Pixar's Toy Story debuted to critical success. For the first time ever, talented engineers and artists were brought together to construct a fully 3d animated feature film. Even more surprising was that the film featured a classic story that would prove to withstand the test of time on par with Walt Disney's pioneering work from the early 1950's. This legitimized the medium. That same year, Dreamworks Animation was created.

In the early days of 3d, gaming purists argued that 3d graphics were a gimmick. The flashy new flat shaded games lacked flair and style FIGURE6.2 . 1993's release of Doom for the PC offered promise, but the pixelated imagery (amazing as it was) was still far from pleasing to the eye. In August of 1996, Id Software's Quake hit store shelves sparking off a series of innovations too long to list here. One month later, in September of 1996, Nintendo released Mario 64 for the Nintendo 64 system proving once and for all that 3d animation had a place in the future of video games.

At this point, 3d animation had been around for some time, but professional grade commercial software was lacking. Without a massive team of talented software engineers, creating world class animation was extremely difficult. To answer this demand, Alias|Wavefront released Maya 1.0 in 1998. Over the next several years, Maya became the definitive tool for 3d animation, especially in the film and television markets

Figure 6.2.



Deformations in a Nutshell

Since it's inception, the Maya software package has included a wide array of deformation tools. Early productions relied on these tools alone to bring their creatures to life. Often times, this meant layering many of Maya's deformers (lattice, sculpt, skincluster, blendshape etc...) to arrive at a final deformation.

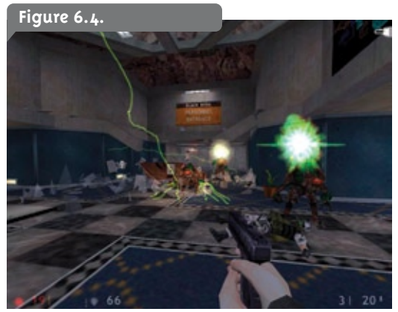
It was not long before the more technically inclined users began to dig deeper into Maya to find new ways of deforming their creatures. Power users consistently rate Maya's extensibility as it's best feature. Ironically, this is the feature that is least used, especially by beginners. Using MEL and the C++ API, third party developers have been constantly developing new methods of deforming creature meshes.

Figure 6.3.

For the video game market, the most widely used deformation method still involves the use of the skincluster and blendshape deformers. Id Software's Quake actually used a method similar to blendshapes to animate all of the creatures in the game FIGURE6.3. All of the animations were stored as a series of blendshapes. A typical animation cycle contained less than twenty shapes that were interpolated at runtime to produce the look of animation. While it was truly three dimensional, the effect was something like a stobe light. The motion was jerky and unrealistic. In 1998, Valve Software's Half-Life for the PC featured the first proper implementation of joint-based animation in a realtime environment FIGURE6.4. Quake was actually supposed to feature integrated inverse kinematics, but the developers (for the most part John Carmack) were unable to integrate the technology in time for the August of 1996 release.

The vast majority of video games now support the combination of both smooth binding to joint hierarchies *and* blendshapes. This has resulted in the ability to sculpt a level of believability previously only dreamed of. While the joints are free to deform the large parts of the body, morph targets (as they are called outside of Maya), can be used to add facial animation and muscle/fat deformation detailing. Recent releases like Half-Life2 and Halo 2 feature engines that allow for this kind of articulation.

Figure 6.4.



As has always been the case, the film and television industries have furthered the development of newer technologies that promise better deformations, easier setup and more control. For the most part, these new technologies are developed as proprietary software plugins and are available only inside the studios that developed them. This breed of deformation tools usually fall into two major categories.

1. Muscle and Skin Simulations
2. Pose Space Deformations (PSD)

When Weta digital was deciding how to deform the creatures they were animating for the popular Lord Of The Rings franchise, they chose to use an in-house solution that simulated the complex interactions between muscles and skin. Many other studios have such software. Tippet Studio's has released some interesting insights into their proprietary muscle system that gained popularity after it's use in the feature film, Hellboy. In November of 2004, CGToolkit released the first commercial muscle plugin for Maya called MuscleTK FIGURE6.5.

It is likely that the future of creature deformations lies in muscle and skin simulations. Muscle and skin simulations promise to bring a level of realism that would be difficult, if not impossible, to achieve using other means. Unfortunately, muscle-based systems are still in the process of being adopted into mid-small size studios and as such, it is still very necessary to be able to build deformations using more 'traditional' means. I use the word 'traditional' in quotations because nothing in 3d animation is truly old or established enough to be called traditional.

Other reasons abound to be well versed in other deformation techniques. Simulation technology still has a long way to go before it is feasible to be used in any real-time applications. For this reason, I would not expect to see video games featuring simulated deformations anytime soon. The other disadvantage of muscle-based deformations is that of control. Because the deformations are simulated, it can be difficult to create custom deformations for very specific shots. This brings me to the current alternative, PSD or Pose Space Deformation.

Pose Space Deformation

Using Maya's default toolset, most riggers will resort to the use of influence objects to help sculpt-out the loss of volume in problem areas like the shoulder or hips. Influence objects are surface geometries that can be added into a skinCluster. The rigger can then paint the influence for these objects, like they would with a joint. The advantage of using a surface as an influence (as opposed to just another joint), is that you can have the actual vertices of the influence object affect the bound skin. For the design of a deformation for a particular pose, the rigger can setup the influence object to translate into a new position based on the rotation value of a joint (this is often done with an expression or set driven keys). Further control can be added by making blendshapes on the influence objects themselves. This setup creates a sort of hierarchy of deformations where the rigger can gain more control over the behavior of the mesh than with the use of joints alone.

There are several disadvantages to using influence objects like this. First of all, influence objects are very slow. By switching-on the use of components, (and thus allowing vertices themselves to affect the skin, as opposed to just transformations of the entire object), the skinCluster is forced to make way too many calculations to provide responsive feedback. The second major disadvantage of this technique is the actual workflow. At no point in time during the setup does the setup artist actually have full and absolute control over how the vertices should behave. This can result is very long setup times and difficult to control deformations. As if these problems were not enough, there really is no way for a real-time application to utilize these sorts of 'hacks' to achieve better deformations. Complex influence objects are completely out of the question for use in current video game engines (and any engines in the foreseeable future).

While many creatures have been successfully rigged using influence objects, most riggers will realize that a more elegant solution would be preferable. Fortunately, this solution exists and it is called Pose Space Deformation or PSD for short.

Figure 6.6.

Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation

J. P. Lewis, Matt Cordner, Nickson Fong

Centropolis

Abstract

Pose space deformation generalizes and improves upon both shape interpolation and common skeleton-driven deformation techniques. This deformation approach proceeds from the observation that several types of deformation can be uniformly represented as mappings from a pose space, defined by either an underlying skeleton or a more abstract system of parameters, to displacements in the object local coordinate frames. Once this uniform representation is identified, previously disparate deformation types can be accomplished within a single unified approach. The advantages of this algorithm include improved expressive power and direct manipulation of the desired shapes per the performance associated with traditional shape interpolation is achievable. Applications applications include animation of facial and body deformation for entertainment, interactive computer gaming, and other applications where fluid sculpting of deformations is desired or where real-time synthesis of a deforming model is required.

while a purely image-based approach can achieve very realistic images, this advantage may be lost if one needs to introduce geometry and surface reflectance in order to re-light characters to match preexisting or dynamically computed environments. Film and entertainment applications require fleshlike creatures that fall outside the scope of image-based approaches.

Some of the most impressive examples of geometry-based (as opposed to image-based) human and creature animation have been obtained in the entertainment industry. These efforts traditionally use shape interpolation for facial animation and a standard but variously-named algorithm that we will term skeleton subspace deformation (SSD) for basic body deformation [25, 9]. While shape interpolation is well-liked by production animators, it is not suitable for skeleton-driven deformation. On the other hand SSD produces distracting defects and is unusually difficult to control.

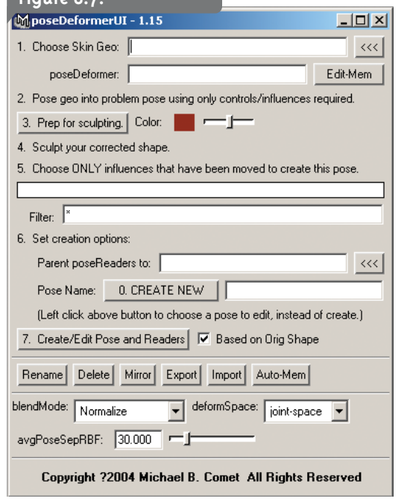
As is often the case with innovation in the field of 3d graphics, PSD was first introduced in the form of a SIGGRAPH paper FIGURE6.6 . The paper is called "Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation". This paper was produced by J.P. Lewis, Matt Cordner and Nickson Fong for the SIGGRAPH conference held in New Orleans in 2000.

Since it's release, this paper has spawned several proprietary software solutions and even one open source project headed by the very talented Michael Comet (www.comet-cartoons.com) FIGURE6.7. While Discreet has incorporated PSD technology into their release of 3ds max 7.0, Alias has yet to follow suit. I suspect Maya will have a native PSD solution soon. More on this soon...

Figure 6.5.

©Copyright 2004 CGToolkit™ Corp. All rights reserved.

Figure 6.7.



So what exactly is PSD, and how does it work? Well, as I mentioned earlier, riggers using Maya's native toolset usually resort to complex 'hacks' that result in too much overhead. As slow and cumbersome as these hacks are, these efforts are necessary to create believable, volume preserving deformations. PSD is a full replacement for influence objects. With PSD, the setup artist adopts a far more artistic approach to generate believable deformations. The actual workflow looks something like this:

1. The artist uses smooth skinning to bind a mesh to a hierarchy of joints. Weighting is adjusted, as usual, to achieve the most realistic looking results.
2. At this point, the smooth skinning alone will leave certain trouble spots. These occur throughout the body, but are especially apparent at ball joints like the hip and shoulder. To fix these, the TD poses the skeleton into a position that creates a nasty deformation, then activates a feature (by pressing a button) of the pose space deformer that allows the problem to be sculpted out.
3. After modeling the mesh into the desired shape, the artist hits another button. The pose space deformer will calculate the world space equivalent corrective shape and connect the interpolation of this shape to the current rotation of the joint. The result being that when you now pose this joint, the deformer will smoothly blend-in the correction.
4. Step three can be repeated as many times as is necessary to ensure that every possible pose results in a nice looking deformation. Underlying anatomical mechanics can be simulated with ease. Bulging, flexing biceps, squishing pectorals and folds of fat can all be sculpted into the skinned mesh to give the creature a very soft, organic quality.

Pose space deformation has several major strengths that make it the sexy new, must-have tool. It is generally quick to setup and easy to use. Pose space deformers act on top of smooth skinning in a very lite way that does not slow down the artistic workflow. Perhaps the best advantage of PSD is the ultimate control that is given over the position of each and every vertex on the mesh. With PSD, setup artists are able to construct deformations that are 'fantastical' or stylized. Unlike simulations that can be difficult to control, PSD enables the TD to create creatures that exhibit deformation styles that run the gamut from photo-realistic to cartoon-esque. There are very few limits.

You may not have realized it, but many of the creatures you see on the silver screen are already using PSD. When ILM was researching creature solutions for the feature film, The Hulk, they finally decided upon PSD. PSD provided the setup artists with enough control over the Hulk's muscular anatomy to provide believable looking skin. The TD's at ILM state that the Hulk required 42 custom shapes just for one of his shoulders! Other parts of the body required less special attention, for example, the Hulk's fingers only needed a couple of shapes to maintain volume. Of course, the amount of shapes needed depends largely on the requirements of the specific shots as well as how large the creature is in-frame. All of these factors need to be taken into consideration when using PSD. Modeling extraneous shapes is a waste of time and should be avoided.

OK, so I've convinced you that PSD is better than chocolate, but one glaring problem remains... Maya doesn't have it! As surprising as this is (considering Alias' keen attention to the needs of creature riggers), there are solutions out there:

PSD in 3ds max: Yes, I know, this book is about Maya, but in case you ever get the chance to work in 3ds max, it may be nice to know that since version 7.0, Discreet has included what they call a Sculpt Morph modifier. This is, essentially, a PSD. Early results indicate that it works very well and incorporates all of the features discussed in the SIGGRAPH paper from 2000.

PSD in SoftimageXSI: Since version 2.0, XSI has featured the ability to use what they call pose-based deforms. This allows a setup artist to hook up blendshapes to the orientation of a joint, or any other object. While this is takes care of additive shape problems (by using a 3d orientation as a driver as opposed to a single-axis rotation value), the user is still left to sculpt the shapes in world space. Something that is extremely difficult to do.

PSD in Animation Master: Oddly enough, this cheaper 3d animation package ($299.00 USD) has featured a PSD-esque feature for some time. Basically, you can pose a joint, sculpt in the correction and save this as a corrective shape. Smartskin, as it is called, is not true PSD, but it is way better than nothing.

PSD in Maya: As of Maya 6.5, Alias has not included any sort of native support for PSD. While this may seem odd (and it is), Alias has probably considered PSD something that individual studios will develop for themselves using the API. While it is true that several proprietary solutions exist, this does not help the end user much because studios rarely sell in-house software to the competition (ie. everyone else). Before you get sad and start sobbing all over this book, I should mention that an open source PS deformer exists and is freely available on the internet! A very talented TD by the name of Michael Comet (M. Comet was a TD at Blue Sky Studios where he worked on the feature animation, *Robots*) has released a plug-in called poseDeformer for Maya 6.0 for MS Windows.

■ Exactly How Awesome PoseDeformerv1.15 Is

As this book goes to print, poseDeformer for Maya is at version 1.15 and already supports all of the important features mentioned in the 2000 SIGGRAPH paper. This plug-in is released under the GNU General Public License and as such, users are free to view the C++ source code and edit it as they wish. To download the latest version of poseDeformer for Maya, visit Michael Comet's homepage at http://www.comet-cartoons.com/toons/melscript.cfm

M. Comet's plug-in is very well built and provides all of the features necessary to get started using PSD on your creature's meshes. In fact, I know of several studios currently using poseDeformer for Maya in a production environment. To find out more about poseDeformer and exactly how to use it, I refer you to the documentation that Comet has prepared, along with some excellent online resources that explain, in detail, how to use poseDeformer to great effect FIGURE6.8. There are several resources that cover, better than I can, all of the ins and outs of using poseDeformer.

Figure 6.8.

Online Wiki: http://www.tokeru.com/twiki/bin/view/Main/MayaRigging
Michael Comet's Hompage: www.comet-cartoons.com
CG Talk Maya Rigging Forum: www.cgtalk.com (go to Alias Maya " Rigging)
Highend3d Maya Forum: www.highend3d.com

Don not be alarmed at all of the reading, poseDeformer is really **very** simple to use (honestly, I'm not lying). After reading through the rest of this chapter, and having used the script-based PSD tool included on the DVD (more about this soon), you will find PSD second nature and *far* easier to grasp than other deformation techniques.

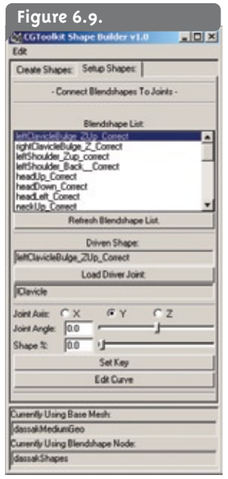
■ So What's the Catch?

As you may have already picked up, I love poseDeformer for Maya. In fact, I get all gushy just thinking about it. But that is not to say that Comet's plug-in does not have it's limitations. Those of you who are reading closely may be wondering about how, if at all, this technology can help games.

The short answer is, not yet. Michael Comet's poseDeformer plug-in for Maya is implemented as a custom deformer that sits in the channel box, right beside a skinCluster node. This is simultaneously it's major strength and weakness. Because it is a custom deformer, it solves quickly and can take care of issues related to additive shapes (by using radial basis functions to generate driver values, more on this later). But, it also means that you can **not** export the mesh to a game engine. In fact, I know of no engines that exist now, or are in development, that are PSD capable. Even the fancy-schmancy Unreal Engine III, Doom III and Half-Life II engines are PSD handicapped.

So rather than sit around whining about it, we at CGToolkit decided to include a little gift in the form of cgTkShapeBuilder.mel (found on the DVD) FIGURE6.9. Shape Builder v1.0 was developed to help TD's in the film *and* game industry implement PSD-like deformations using nothing but regular blendshapes (and some clever trickery). Using Shape Builder, setup artists can quickly sculpt fixes in pose-space with full control over interpolation (via set driven key curves). In addition, users may wish to use Shape Builder outside of a game development environment simply because no additional software is needed to read the resulting rig files. Comet's poseDeformer plug-in requires that either the entire rendering pipeline has poseDeformer installed, or the mesh is baked out using a proprietary mesh baking solution (like CGToolkit's Shot Sculptor. Maya's mesh baking feature is simply not fast enough). Render farms with stock versions of Maya will have no problems reading files rigged with Shape Builder because the resulting deformations are attained with the use of the regular Maya blendshape node (no custom deformers must be supported).

Figure 6.9.



Both poseDeformer and Shape Builder have their advantages. It will be up to the needs of the specific production to decide which to use. As an added bonus, we include complete coverage of how cgTkShapeBuilder.mel was designed (found at the end of this chapter). This detailed explanation will have three major uses:

1. As a nice introduction to matrix math and how to code it in MEL.
2. To give you a better understanding of coordinate space transformations (a fundamental concept in 3d math and programming).
3. To help you learn how to write better MEL scripts. The Shape Builder MEL script example includes full coverage of error checking techniques (something we have been ignoring up to now).

As with all programming problems, it is best to have a *very* innate understanding of exactly what the problem is before attempting to create a solution. It may sound obvious, but most failed development projects can have their demise traced back to a lack of fundamental understanding about the exact nature of the task to be solved.

With this in mind, consider the rest of this chapter to be both a complete coverage of PSD techniques (including artistic considerations) as well as preparation for the MEL lesson at the end of this chapter.

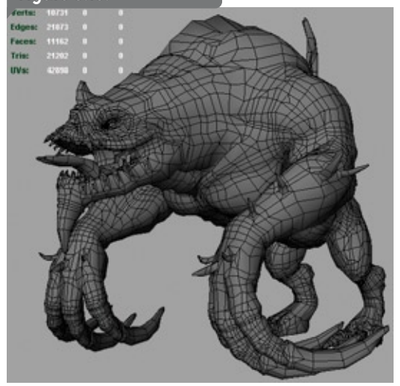
In fact, using the techniques presented in this chapter, you can build a creature rig with realistic flesh bulging, muscle-to-muscle collisions, volume preservation and even dynamic fat jiggle. Even more exciting is that we will show you how to do this quickly and efficiently and in a way that uses only joints and blendshapes (so you can use your creatures in a game engine!). All of this is done with no sacrifice in quality. Personally, I actually prefer these techniques for use in film resolution deformations as well. So, without further ado, let's get started!

Exercise 6.1 Using Shape Builder v1.0 on Dassak's Knee

Rather than get bogged down in some of the more theory intensive material, why don't we go through a full example right of the bat? In this exercise, we will be using CGToolkit's Shape Builder script (cgTkShapeBuilder.mel) to setup a calve-to-thigh flesh collision.

Our model for this chapter is a hulking beast by the name of Dassak. For those interested, Dassak was modeled in Pixologic's Zbrush v2.0 before being exported to Maya in .obj format FIGURE6.10.

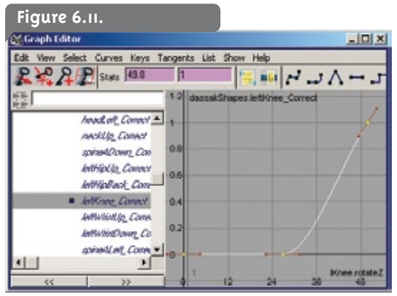
Figure 6.10.



The workflow I used to setup Dassak was very straightforward. To setup a creature using Shape Builder deformations, I recommend following a general series of steps. This will help you avoid inefficiencies that can crop up from poor preparation.

1. Check the integrity of the mesh. This is quite standard but very important. Proper mesh construction is especially important for this kind of deformation technique. If a problem is found in the base mesh, changes to the topology can render your hard work unusable. Remember, Shape Builder uses regular blendshapes and as such, is extremely picky about vertex ordering.
2. For Dassak, I laid out the joint hierarchy for the entire left side of his body first. After naming and meticulously orienting the joints by hand (by editing local rotation axis), I used the `Skeleton>Mirror Joints` tool. This ensures that joint orientations are symmetrical. This is especially useful for rotating corresponding left and right joints at the same time to observe any asymmetric deformation problems. Mirrored orientations will allow animation from the left side to be copied to the right. This is essential for game creatures that have a lot of looping animation cycles.
3. To bind Dassak, I used regular Maya smooth skinning. Artisan was a great help to flesh out the weighting on the left side of the body and `Skin > Edit Smooth Skin > Mirror Skin Weights` was used to copy weighting work across the YZ plane.
4. It is important to get the weighting as good as possible before you start sculpting corrections. Sculpting a correction that could have been fixed with weighting is a waste of time and effort.
5. After finalizing and testing the weighting, you can begin using Shape Builder. For Dassak, I went through each joint in his body and hand modeled corrections at each extreme pose. After modeling the corrections, I saved the corrected blendshape (more on this soon) into a separate layer.
6. To mirror these shapes across the YZ plane (and thus cut your sculpting time in half), you can use any number of scripts available on the Internet. I use abSymMesh.mel, found on highend3d.com. Full coverage of abSymMesh is included on page 116.
7. With all of the finished, corrected and mirrored shapes, you can begin connecting the shapes to joint rotations. Shape builder provides a fast way to connect shapes to joints via set driven keys. To provide extra control over the exact interpolation of the shape, you can edit the set driven key curve itself FIGURE6.11.

Figure 6.11.

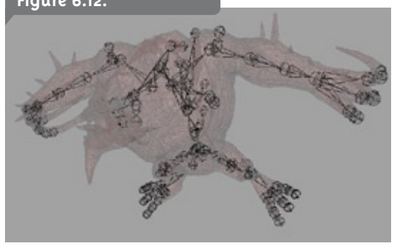


This exercise is a just a snippet of the workflow from setup five above. The knee is a common problem area that requires some special attention. Because the knee only rotates in one axis (simple hinge joint), the knee is a perfect place to start our explanation of Shaper Builder.

Copy cgTkShapeBuilder.mel from the included DVD. Place this script file in your maya/scripts directory, restart Maya and type "cgTkShapeBuilder;" into the command line to bring up the Shape Builder UI.

1. Open exercise6.1_Start.mb. This file contains Dassak's rig, post-weighting. Notice that there is no iconic representation or high-level controls of any kind. I prefer to work on a pure FK skeleton while setting up corrective shapes. This allows easy, direct manipulation of joints to observe the affects of your work. If you go ahead and grab some of his joints now, you will notice that the left/right sides of his body behave differently FIGURE6.12. In this scene file, Dassak's left-hand side is completely rigged with corrective shapes that were connected using Shape Builder's set driven key feature. The right hand side is deformed with basic smooth skinning alone.

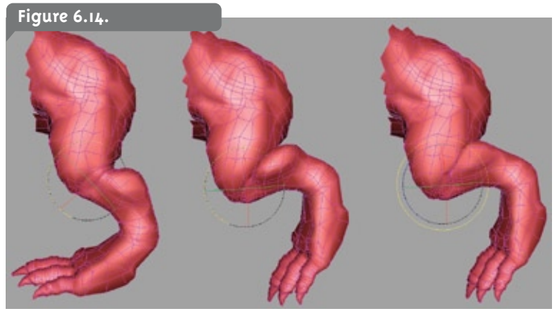
Figure 6.12.



2. In wireframe mode, grab Dassak's left and right clavicle joints. Rotating these upwards in Z is a good way to see the exact effect that the corrective shapes are having on his mesh. In the case of the clavicle, I created a shape that causes the trapezius muscles to compress, flex and bulge. You will also notice that the right side has several inter-penetration problems FIGURE6.13 . These were easily remedied on the left side with a corrective shape. Play with Dassak's skeleton and observe the differences between the left (corrected) and right (non-corrected) deformations.

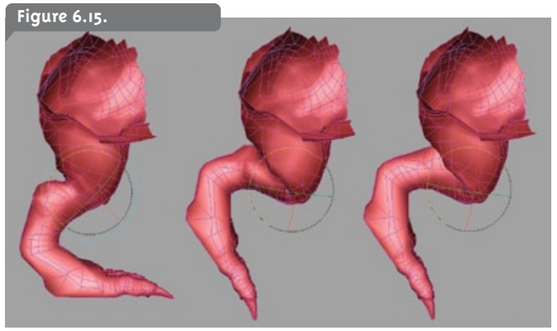
Figure 6.13.

Figure 6.14.



3. If you haven't already, grab the left knee joint and rotate this upwards. Notice that as the calve is about to collide with the thigh, the mesh bulges outwards and maintains volume. Also notice how the shape appears to interpolate in a very non-linear fashion, this creates the illusion of a muscle-to-muscle collision. The bulging does not occur until the calve and thigh are almost penetrating. If you compare this behavior to the right knee, you will immediately notice the right knee has some problems. In FIGURE6.14 you can see the knee in it's default pose, the left side with bulging and volume preservation, and finally the right knee with no corrective shapes at all. In FIGURE6.15 you can see this from the inside of the leg. The right knee joint can rotate to about positive 25 degrees while looking fine, but after this, the calve collides right through the thigh. This is the default behavior from smooth skinning. No amount of weight adjustment can resolve this.

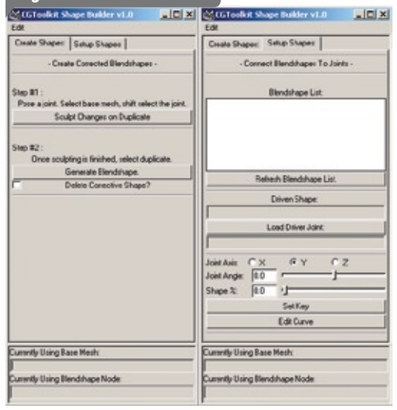
Figure 6.15.



4. To fix this, we are going to sculpt a correction for the right knee and hook it up to the Z axis of the knee joint. Select any one of Dassak's joints and select `Skin > Go to Bind Pose`. It is important to be in the bind pose before attempting to sculpt a corrective shape. If a joint is rotated out of it's default pose, unbeknownst to you, the shape may not work as you expected. This is another reason why I like to work on a pure FK rig while doing corrective shapes. IK can prevent a mesh from being able to go back to it's bind pose.

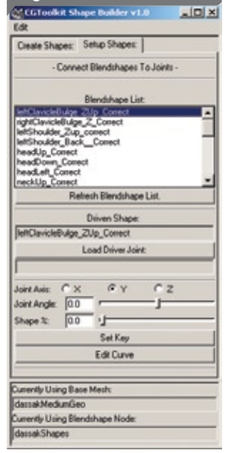
5. Now, with Dassak in his bind pose, we are ready to get to work. Open Shape Builder ("cgTkShapeBuilder;" from the command line, or make a shelf button) and take a look at the interface. The first tab, 'Create Shapes', is used to duplicate and generate meshes to create your corrective shapes. Remember, these are just regular blendshapes. The second tab, 'Setup Shapes', includes all of the tools needed to quickly setup set driven key connections between the rotation values of your joints, and the blendshape weight (ie. strength) FIGURE6.16.

Figure 6.16.



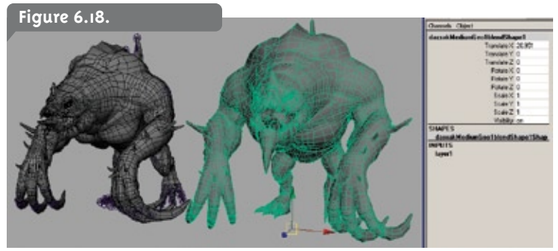
6. Shape Builder requires some information about the current mesh you are using. This is necessary to use the automated set driven keys. Setting up Shape Builder is easy. First, select the mesh you are working on (in this case Dassak) and choose `Edit > Load Base Mesh`. This will put the name of the base mesh into the greyed-out text box at the bottom of the interface. Lastly, we need to load the blendshape node. With Dassak selected, open the channel box and click on his blendshape node (named dassakShapes). With this node selected in the channel box, click `Edit > Load Active Blendshape Node`. This is all the setup needed to use all of Shape Builder's features FIGURE 6.17.

Figure 6.17.



7. Grab the knee joint and rotate it to about positive 48 degrees (in Z). This will create a condition where the calve and thigh are quite ugly looking. In this pose, we have a severe loss of volume and nasty inter-penetration. To start sculpting the fix, select Dassak's mesh, shift select the knee joint and click on the 'Sculpt Changes on Duplicate' button in the 'Create Shapes' tab of the Shape Builder interface FIGURE6.18.

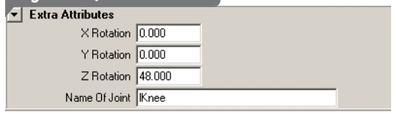
Figure 6.18.



8. This will result in a proper duplicate mesh that is generated and moved off to the side. This is the mesh we will be sculpting on. Select this mesh (it should be named, 'dassakMediumGeo1blendShape 1'). Rename this mesh 'rightCalveFix' or something meaningful. If we were to sculpt our fixes and add this shape into the current blendshape node as it is, we would get a shape that affects the entire leg. This is the main reason why a script like Shape Builder is necessary. Without it, we cannot sculpt the fix in pose space. With Shape Builder, you can sculpt the fixes in pose-space and the script will take care of subtracting the initial deformation (from the knee joint) to ensure that the shape behaves as we need it to.

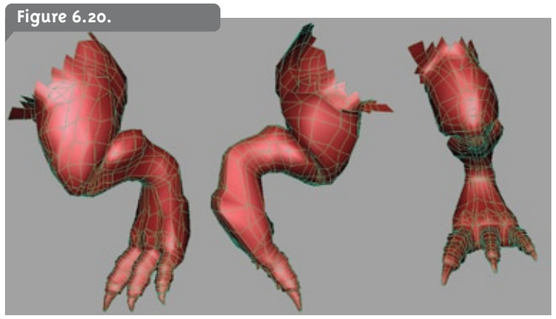
9. You may be wondering why you had to select a joint before using the duplicate button. This is done so that the script can record the exact pose that the shape was created at. To see what pose was recorded into the duplicate, select the mesh and open the attribute editor. Under the transform node, in the 'Extra Attributes' tab, there should be four new attributes. The script records the name of the selected joint and value of it's rotations FIGURE6.19. This is not absolutely necessary (the script will work fine if you leave the model in the initial pose) but it can prevent errors that will occur if you rotated the joint after duplicating the mesh.

Figure 6.19.



10. OK, that was a lot to digest, but this all becomes very clear after a couple trials. Trust me. At this point, we can sculpt the fix. I use a combination of soft modifications, the sculpt polygon tool (with smooth and push/pull operations) and manual point pushing. Sculpt the shape into something like FIGURE6.20. By pulling the the flesh outwards, we can maintain volume. You may find it handy to drop the camera inside the thigh and use the sculpt polygons (pushing) artisan brush to push the calve out of the thigh. Maya's modeling tools are great for quickly sculpting these kinds of shapes. You may be wondering how to tell exactly how much squishing you should add to the edges of the calve and thigh. This is sort of an artistic decision. My only advice is that you trial and error the shape into perfection. This can be done very quickly using the Shape Builder's 'Setup' tab.

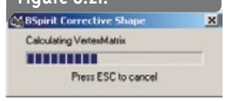
Figure 6.20.



11. With the fix sculpted, we are almost finished. Select the mesh and click 'Generate Blendshape'. This will bring up Bspirit's (more on Bspirit later in this chapter) progress window FIGURE6.21. For Dassaks mesh (10 507 vertices) I found that shape genera

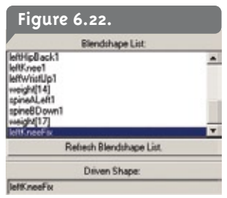
tion takes only a couple seconds (P4 2.8GHZ, 1gb). Extremely high resolution meshes may take up to several minutes to compute. Fortunately, this only has to be done once and after the shape is generated, it performs as quickly as a regular blendshape.

Figure 6.21.



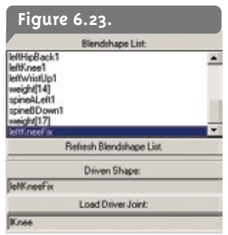
12. This is all that the 'Create Shapes' tab does. Duplicate mesh, generate shape, repeat. Of course, we are not finished yet. To get this shape hooked up to the rotation of the knee joint, we will use the other tab in Shape Builder, 'Setup Shapes'. Click on this tab and find the 'Refresh Blendshape List' button. This list contains all of the shapes in the currently loaded blendshape node (recall that we loaded a blendshape node in step 6). When we generated the knee shape, it was automatically added into the loaded node and will appear in this list after hitting the 'Refresh Blendshape List' button. Clicking on the shape will load it into the 'Driven Shape' text box FIGURE6.22. We are now ready to start setting this shape up.

Figure 6.22.



13. Select the right knee joint and click 'Load Driver Joint' to load it into the script FIGURE 6.23. Shape Builder will only setup shapes on joints. If, for some reason, you need to connect the shape to something else, you will need to use Maya's native set driven key tool.

Figure 6.23.



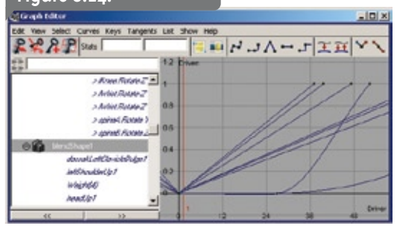
14. Select the 'Z' radio button to set this as the joint axis. This is the axis the knee bends on. Notice that scrubbing the joint angle slider will rotate the knee joint in the viewport. When you loaded the joint, the script connected the joint to this slider for quick editing. Similarly, the 'Shape %' slider will control the weight of the currently loaded blendshape.

15. With the joint angle and the shape percentage set to '0'. Hit the 'Set Key' button. Recall that we sculpted the correction when the knee joint was at 48 degrees. Set the joint angle to 48 and the shape percentage to 100. Hit the 'Set Key' button to set a final key.

16. If you now rotate the knee joint (or use the joint angle slider in the script), you will see the calve-fix blend-in smoothly. While this, in itself, looks pretty cool, we can enhance the effect by adjusting the interpolation. In the case of the calve, we want the shape to remain 'off' until the flesh is about to collide. It would be nice to have the shape overload if the knee is posed past 48 degrees. This can avoid the odd jarring effect where the shape suddenly stops interpolating.

17. To adjust the interpolation, we will be editing the animation curve that is created from the set driven keys. Shape builder provides a quick way to access this. Click the 'Edit Curve' button to bring up the graph editor with the blendshape automatically loaded FIGURE6.24.

Figure 6.24.



18. Scroll through the shape list in the graph editor to find the knee fix shape. With this selected, hit the 'f' key to frame the selection around the curve. At this point we have a linear curve with only two keys FIGURE6.25. This causes the shape to blend in a straight fashion. We want the shape to stay 'off' and kick in where the calve begins to collide with the thigh (at about 25 degrees). Editing the set driven key curve is the same as any other animation curve. Add a key at 25 degrees and move it downwards. Select all of the keys and give them flat tangents FIGURE6.26. This will add a slight ease-in/out. To prevent the shape from cutting off after the last key, select the curve and execute `Curves > Pre Infinity > Linear`. This will cause the shape to continue blending, even when the maximum pose has been reached. Use this with caution because this will overload the blendshape and may result in odd looking deformations. But posing the knee past 48 degrees is unlikely and unnatural anyway.

19. If you were setting up the rest of the body, you could continue on in this fashion. Once you get going, this workflow of sculpting and setting up fixes is far faster and much more efficient than any alternatives.

20. Open exercise6.1_Finished.mb to see Dassak with all of his shapes connected. Try posing his joints to see how it all works.

Figure 6.25.

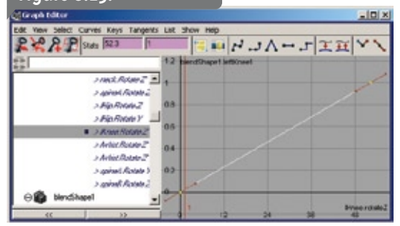
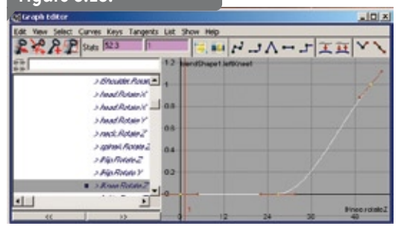


Figure 6.26.



That completes the introduction to Shape Builder. While setting up your own creatures, pay special attention to how you setup interpolation. For the most part, common sense will dictate how a shape should setup. While sculpting the mesh fixes, keep in mind the principles of squash and stretch, volume preservation and muscle contractions.

Once you are feeling brave enough, try taking these ideas a step further. You can get as detailed as you wish with corrective shapes. In fact, you can sculpt corrections on-top of other corrections (the script will work through any deformers). This might be the beginnings of an advanced combination-shape facial setup.

On the interpolation side, there are lots of possibilities. Set driven key curves can be edited to your heart's content. To give a muscle a very slight vibration (indicative of being under stress), you could simply add a ton of keys in the graph editor. Experimentation will undoubtedly yield some interesting effects.

As you experiment further with interpolating shapes based on joint angles, you will happen upon a limitation of Shape Builder. The problem rears it's ugly head when trying to setup shapes on multiple axes at once. This is most common in areas of the body with ball joints. Hips and shoulders, unfortunately, are especially susceptible. This brings me to the next topic of discussion.

■ Defining Complex Poses

This section is here to help you avoid a common pitfall encountered when using pose space deformations and blendshapes. Often times, set driven key connections can result in a situation where blendshapes driven by multiple axes are added together. This problem can be minimized by constructing corrective shapes with non-overlapping areas of influence. This is a perfectly viable solution for most situations, but in some, more extreme cases, more control is needed.

When you use Shape Builder's setup feature (or just Maya's native SDK tool), you are creating a connection between two scalar values. If you recall from math class, a scalar value is one with a *single element of data*. That's a fancy way of saying that a scalar value is just a single number.

By using a single scalar value as a driver for a blendshape, you run the risk of not 'defining' exactly when you want the blendshape activated, in an accurate way. This is a loss of control issue that is directly related to the idea of what a pose actually is. If this does not make sense yet, please hang in there.

To understand exactly what causes this problem, we need to discuss the three different types of joints. Understand that these solutions are somewhat geared towards more advanced setups. It may be that set driven keys provide all of the control you will ever need. Especially for game creatures that are unlikely to be hitting any extreme poses (that would necessitate a more accurate definition of a pose).

■ Single Axis Drivers

Recall that we want to be able to activate a corrective shape for our creatures at several specific poses. In exercise 6.1 (please read exercise 6.1 now if you have not already), we defined the extreme pose for the knee as the point at which the knee joint's Z axis was rotated to positive 48 degrees. For the knee joint, we defined the 'target' pose in terms of the rotation value of a single axis (Z). This worked perfectly fine.

In fact, hinge joints (joints that have a single axis) only require a single scalar value to define a pose for. Hinge joints are all over the body. The elbow, knuckles (second and third), toes and knees are all prime examples of hinge joints. In fact, for these parts of the body, set driven keys will be all the control you will ever need.

■ Twin Axis Drivers

Joints that must be defined in terms of two-axes (like Y and Z) present an interesting challenge. Unlike a hinge joint, a scalar value will no longer accurately define the pose. Examples of these types of joint's include wrists, ankles and first knuckles. Basically, these joints can rotate in any direction, but lack the twisting motion provided by a ball and socket.

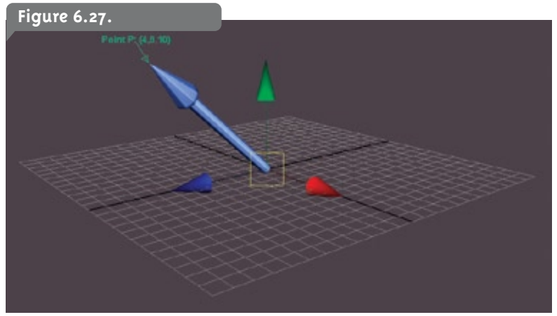
So the obvious question is, how do we drive deformations on a joint who's pose cannot be defined by a scalar? There is no definite answer to this question. While the solution I am going to present in the next section is certainly capable of defining a pose for this type of joint, it is likely overkill. In my experience, twin axes joints usually only require a couple of corrective shapes who's areas of influence barely overlap. For joint's of this type, set driven keys are probably fine. Of course, cases will arise where more control may be needed. For these cases, it can help to have an understanding of exactly how to define a pose for a twin-axes joint.

To accurately define a pose for a joint of this type, you must use three scalar quantities. Three scalar's are also known as a vector (or, more specifically, a 3d vector). A vector is nothing magic, it's just a way of grouping related scalar values. But why, you may ask, do we need a three dimensional vector when the joint only has two axes?

To help you see why a 3d vector is needed, let's have a quick discussion about vectors in general. Consider the vector P, defined as [4,8,10]. There are many way's to interpret P, dependent on the context in which

it was created. P could define a Cartesian coordinate. If this were the case, P would define a point in 3d space where X=4, Y=8, and Z=10. What is more interesting, is what P can define when we compare it to the origin (the point 0,0,0). Drawing a line from the origin to P, we can easily see that P defines not only a point in space, but a direction as well FIGURE6.27. Quite simply, this is the direction our joint is pointing and thus, our pose.

Figure 6.27.



When I try to visualize a vector (in my mind's eye), I see it as a series of axially aligned displacements. The vector [4,8,10] is actually a displacement of 4 units in X, then 8 units in Y, followed by 10 in Z. Thinking of vectors like this can help them make more sense.

If you find that you need to use a more accurate driver for a twin axis joint, you can use a multi-axes driver expression (more on this soon) with the twisting disabled. This has the effect of defining the pose with a single 3d vector.

■ Three-Axes Drivers

Alright, this is the real deal. As you may have already guessed, joints that fall under this category are of the ball and socket type. The most notorious of which are the dreaded shoulder and hip joints. It seems like these areas of the body were invented, by god, as a cruel joke for the sole purpose of annoying setup artists across the globe. The main problem that arises from these joints is that of (you guessed it) the third axis. This is the twisting motion, without which, you would find it very difficult to swim, golf, or throw a baseball.

Defining a pose for this type of joint requires a combination of a 3d vector *and* a scalar. Just like with the twin-axes joints, the vector describes the direction the joint is pointing in. The newly added scalar value is used to define the *twisting* motion. When combined, the directional vector and the twisting value leave us with a complete pose definition. For example, a shoulder joint's target pose might be described as the directional vector [4,8,10] plus a twisting value of 30 degrees (the scalar).

■ The Multi-Axis Driver Expression

Recall that the whole point of defining poses is so that we can drive a blendshape based on how similar the current pose is, compared to the target pose. To help visualize, let's consider a real life scenario. Imagine a creature has been fully setup using multi-axis drivers. A pose has been defined for the creature's shoulder where it is rotated upwards *and twisted* slightly FIGURE6.28. At this pose, a correction was modeled to fix the arm deformations. Now, as the animator poses the shoulder straight upwards FIGURE6.29, the shape remains off (as we want). Even as the shoulder's direction approaches the target pose, a multi-axis driver should not start increasing the blendshape weight. Only as the animator begins twisting the shoulder (and thus reaching the target pose) does the driver increase the blendshape weight to blend the fix.

Figure 6.28.

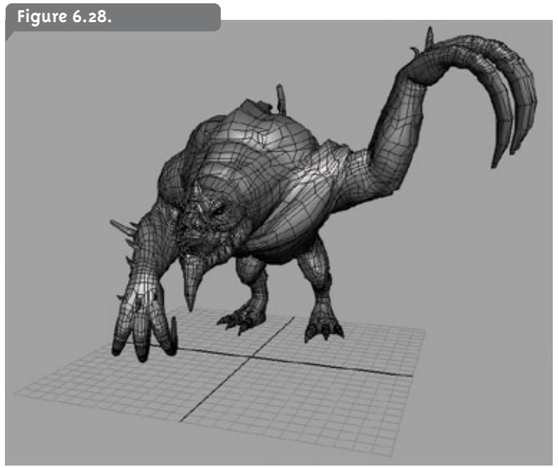
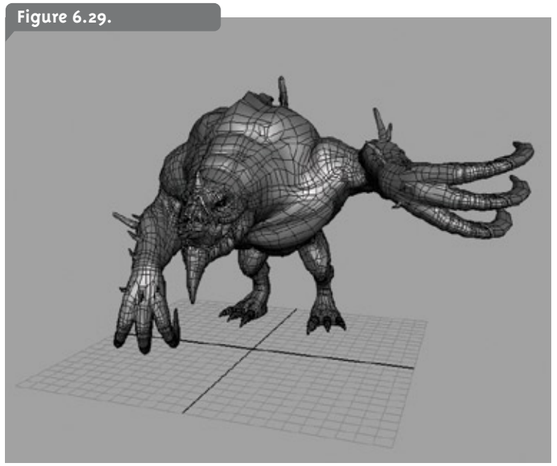


Figure 6.29.



This kind of driver behavior allows us to sculpt many, super-specific fixes and have them blend in only at certain, precise poses. In practice, the solution presented in this section takes the form of a MEL expression. To setup a creature using this expression, it is as simple as adding a new expression for each specific pose, tuning some parameters and hooking up the returned $outputWeight (a float value normalized from 0 to 1) value to a blendshape weight.

To get started, let me first describe the process needed to setup the expression. As I mentioned earlier, the expression needs some inputs. If you copy the contents of multiAxisDriver.mel (included on the DVD) into the expression editor, you will see two large commented sections that encompass the setup variables.

```
////////////////////////////////////////////////////////////////////////////
//              - CGToolkit's Muti-Axis Driver Expression -
//
//      To use: Setup variables below. Create expression.
//
//This script returns a float variable named $outputWeight.
//This value represents how 'similar' the current and target
//pose are. It is normalized between 0 and 1 for easy hook-up
//to a blendshape. Scroll down to the bottom for instructions on
//hooking this up.
////////////////////////////////////////////////////////////////////////////

//Direction of Pose. Setup-Variables

//The name of the locator constrained to the driver joint.
string $nameOfDriverLocator = "leftArmVector";
//The name of the locator in the target position.
string $nameOfTargetLocator = "targetVector";
//Max angle (degrees) deviation before weight = 0 (shape off).
float $maximumAngle = 55;
//Min anlge (degrees) where weight will stop increasing.
float $minimumAngle = 0.00;

//Twisting. Setup-Variables

int $enableTwisting = 0 ; // Turn twisting on/off. 1=On, 0=Off
//Angle at which twisting should start. (Ususally zero)
float $startTwist = 0.00;
//Amount of twisting in the target pose.
float $targetTwist = -60.00;
//Name of the joint that is twisting.
string $nameOfTwistingJoint = "shoulder";
// Axis that joint is twisting in. By default, 'x'.
string $twistAxis = "x";

////////////////////////////////////////////////////////////////////////////
//                    SETUP COMPLETE
//                EDIT AT YOUR OWN RISK
////////////////////////////////////////////////////////////////////////////
...
```

As you can see, the setup is not that bad. There are four variables needed to define the direction of the joint, and an additional five used to control the twisting. Setting them up is pretty straightforward, but there are some procedures you must follow.

1. $nameOfDriverLocator: This is the only tricky part of the expression setup. This variable is what is ultimately used to track the vector for the current pose. To do this, use a locator (or any other transform node, like a null) to follow your driver joint. This is important, *the locator's translates will be read as the joint's current vector*. To ensure that the locator's translates contain the proper information, you must follow a couple of steps. Create the locator and snap it to the driver joint (ie a shoulder or hip). Freeze transforms on the locator and then *snap it to the child of the driver*. Now the translate channels will contain the vector from the driver joint to it's child. This is, effectively, the direction the joint is pointing in (a joint always points to it's child). To ensure that the locator's translates are updated during animation, *point constrain the locator to the child joint* (ie. the elbow or knee joints). If you used a locator named 'shoulderDriverLocator', you would edit this first line of the expression to read 'string $nameOfDriverLocator = "shoulderDriver-Locator";. Now the expression can track the driver joint's direction.

2. $nameOfTargetLocator: Repeat step one but this time snap the locator to the child joint *in the target pose*. This locator will house the target pose vector. Because the target pose vector will change if a parent joint is rotated (like a clavicle), you need to parent constrain this locator to the parent joint (for a shoulder driver, this locator must be constrained to the clavicle).

3. $maximumAngle: This value controls the interpolation of the shape. Larger values cause the shape to blend-in quicker. This value is a measurement of the angle between the current pose and the target (measured in degrees). Try starting at 50 degrees and tweak it from there.

4. $minimumAngle: The minimum angle specifies the angle (in degrees) at which the shape will stop increasing. This can be used to 'cap' a blendshape so that it will not reach it's full strength. Keep this at '0' unless you have a special case that requires it to be raised.

5. $enableTwisting: This attribute will turn on/off the twisting feature. A value of 1 will cause the expression to return a weight that factors-in the similarity of the twisting axis (by default the x axis).

6. $startTwist: The angle (in degrees) at which the twisting begins. Leave this at 0 unless your joint's x axis is not 0 in the default pose. Adjusting this value can have the effect of adjusting the interpolation of the twisting axis.

7. $targetTwist: This is the value of the twist in the target pose. This number can be read right out of the channel box as the 'rotateX' value from your driver joint when it is in the target pose.

8. $nameOfTwistingJoint: Feed this variable the name of the driver joint (ie. "shoulder" or "hip").

9. $twistAxis: Some people prefer to orient their joints such that any axis could be pointing down the bone. By default, Maya orients joints such that 'X' is the twist axis. Just be sure that this value is lowercase or the expression will return an error.

After declaring all of the setup variables, the last task is to connect the actual blendshape. To do this, scroll down to the bottom of the script until you see this:

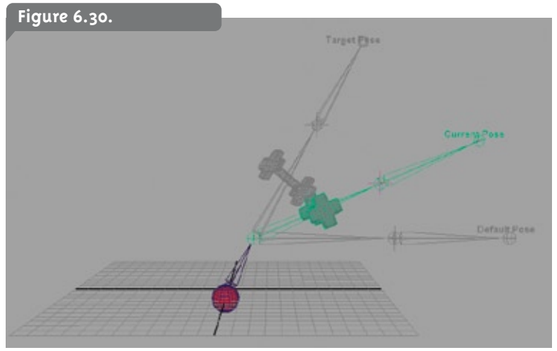
```
////////////////////////////////////////////////////////////////////////////
//                CONNECT DRIVEN ATTRIBUTE HERE
////////////////////////////////////////////////////////////////////////////

//This is how you could now hook this up to a blendshape.
//Edit and uncomment the following line.
//nameOfBlendshapeNode.nameOfDrivenShape = $outputWeight;
```

The last line above demonstrates how the expression can be hooked up to a blendshape. The $outputWeight value was designed to be hooked up to a blendshape and is normalized from 0 to 1. To use it, uncomment this last line and edit the `nameOfBlendshapeNode.nameOfDrivenShape` to whatever you want.

To see this expression in action, open the example scene files in the included DVD (scene files). These files contain a sample shoulder setup. The red sphere beside the joint chain has had it's shader connected to

expression to represent the strength of the weight. When the weight is 0, the sphere is red. As the arm joints get closer to the target pose, the sphere will gradually turn green to show the strength of the weight increasing. Playing-back the animation shows how the expression determines the similarity of the current and target poses FIGURE 6.30.

Figure 6.30.



For those interested, these scene files contain versions of the expression with added commentary that explains exactly what is going on. If you watch the command feedback line, you will notice the script prints out the current weight as you scrub through the animation. Opening the script editor reveals extra information about exactly what the expression is calculating FIGURE6.31.

Figure 6.31.

The actual guts of the expression make heavy use of MEL's excellent vector maths functions:

```
//Declaraction of Variables
vector $currentPoseVector; //Computed at run-time
vector $targetPoseVector; //Computed at run-time
vector $targetNormal; //Normalized version of target vector
vector $currentNormal; //Normalized version of current vector
float $angleBtw; //Angle (rad) between target/current vectors.
float $outputWeight; //Output that will drive blendshapes.

//Find current and target coordinate vector
float $currentRotationValues[] = `getAttr ($nameOfDriverLocator
                                          + ".translate")`;
float $targetRotationValues[] = `getAttr ($nameOfTargetLocator
                                          + ".translate")`;

//Fill the $currentPoseVector, and $targetPoseVector
$currentPoseVector = <<$currentRotationValues[0],$currentRotatio
nValues[1],$currentRotationValues[2]>>;

$targetPoseVector = <<$targetRotationValues[0],$targetRotationVa
lues[1],$targetRotationValues[2]>>;

//Normalize the target and current vectors
$targetNormal = unit($targetPoseVector);

$currentNormal = unit($currentPoseVector);

//Calculate AngleBtw
float $dotProduct = dot($targetNormal, $currentNormal);

if ($dotProduct) //If $dotProduct = 1, angle is 0
{
        $angleBtw = 0;
} else
{
        $angleBtw = acos($dotProduct);
        $angleBtw = rad_to_deg($angleBtw);
        //Calculate the Weight
        $angleBtw = clamp ($minimumAngle, $maximumAngle,
                                        $angleBtw); //Angle calulation
        //Normalized Weight from Angle
        float $outputWeight=abs(($angleBtw / $maximumAngle) -1);
}

//Calculate the Weight
$angleBtw = clamp ($minimumAngle, $maximumAngle, $angleBtw);
//Angle calulation
//Normalized Weight from Angle
float $outputWeight = abs(($angleBtw / $maximumAngle) -1);

//Factor in Twisting
if ($enableTwisting)
{
        float $currentTwist = `getAttr ($nameOfTwistingJoint
                                        + ".r" + $twistAxis)`;
        float $rangeOfTwisting = abs($targetTwist - $startTwist);
        float $twistFactor = ($currentTwist / $rangeOfTwisting);
        $outputWeight = abs($outputWeight * $twistFactor);
}
```

The first half of the expression is designed to gather the required variables and values. Actual computation begins with normalizing the target and current pose vectors. Normalizing a vector has the same effect as cutting it's length down to 1. Changing the length of a vector (also known as the magnitude) does not change the direction it points in.

With the vector's normalized, we can calculate the angle between them (you could do this without normalizing them, but the equation is far more complex and susceptible to division by zero). The angle between two vectors (as measured in the plane they both lie in) is related to the dot product of the two vectors. To calculate the angle in radians (between the target and current pose) we can take the dot product of the two vectors and multiply this by the arc cosine (noted acos).

```
$angleBtw = clamp ($minimumAngle, $maximumAngle, $angleBtw);
```

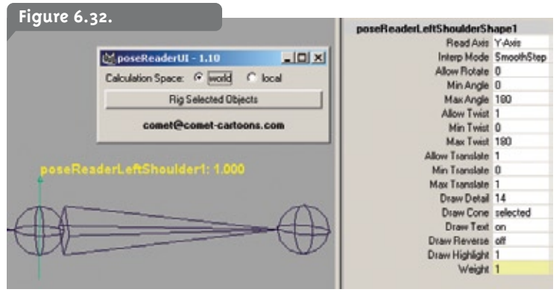
This leaves $angleBtw storing the current angle (in radians) between the target and current poses. Because we are working with degrees, the next line converts this angle from radians using the rad_to_deg MEL function. After normalizing this value between 0 and 1 (remember blendshapes go from 0 to 1), we need to factor-in the twisting.

The 'if ($enableTwisting)' line checks if the user has specified twisting to be enable/disabled. If $enableTwisting is equal to something other than zero, the script calculates the normalized ratio of twisting (as compared to the $targetTwist value) and factors this into the weight.

That's all there is to this little expression. The real strength of this script lies in it's re-usability. Because the script uses universal methods of defining and comparing poses, it can be easily adapted to any rig. I have seen other expressions that try to accomplish a similar effect in a top-down approach. This becomes extremely troublesome as the code quickly devolves into massive amounts of if or cases statements. The top-down approach is difficult to setup, not easily reusable and heavier to calculate.

Also important to note is that Michael Comet (the author of poseDeformer for Maya) has released a standalone plugin that mimics this expression exactly. The plugin is called poseReader and as this book goes to print, the current version provides some features that this expression does not. Most importantly, Comet's plug-in provides a nice visual feedback of the maximum angle as well as some better interpolation methods (including smoothstep, gaussian, linear or even a custom animation curve) FIGURE6.32. Like all of Michael Comet's treats, poseReader can be downloaded from www.comet-cartoons.com. User's running something other than Windows may need to recompile from the included source code (or use the expression) because Comet only provides Windows binaries.

Figure 6.32.



Regardless of whether you use a custom plug-in or the multi-axis expression, understanding how pose drivers can give you more control over your deformations will give your creatures that extra polish that separates them from the mainstream.

Speaking of polish, this chapter is almost ready to depart from the technical, nerdy, stuff. After the next section, I will discuss some of the more artistic considerations in deformation sculpting. Before that though, I want to ensure that the reader has a complete understanding of exactly what is going one when they create a 'corrective' blendshape.

■ Beautiful, Lovely, Sexy Corrective Shapes

Personally, I hate working with software when I have absolutely no idea how it works. Even having a very basic idea of the underlying algorithms can, often times, provide you with new and better ways to optimize your workflow and generally improve your work. While this is true of most applications, it is especially true of 3d animation software.

In the case of rigging, for example, understanding how the different deformers work allows you to adjust your workflow accordingly. A prime example of this is the skinCluster deformer. Without knowing that each vertex on a mesh must have it's smooth skin weight normalized to a value of one, the skinCluster's behavior would seem odd and even buggy. In fact, when I first started rigging in 3d, I found smooth skinning to be a very annoying, buggy mess. But when someone finally sat me down and explained it, a light bulb went off. Since then, weight painting has become a very natural, even artistic, workflow for me.

In this chapter, I have introduced you to a new sort of deformation, PSD. While the script-based technique used in this book does not actually use a proper deformer, it does use an algorithm that affects the points on your sculpted mesh in the same way that a deformer does.

When I was researching for Shape Builder v1.0, I knew, in my mind, exactly what I wanted it to do, but I needed a place to start. Before a software developer decides to green-light a project, they do some market research to determine what the competing products are. In my case, I just wanted to see if anything existed that could potentially lessen the amount of work I would have to do to get this thing built (and ready for you, the reader, to use). So I started at the usual Internet forums and within two days I became aware of a similar script written by a friendly chap from Germany by the name of Christian Breitling. Christian's script is excellently constructed and provides the core functionality for Shape Builder. His script (available on highend3d.com) is called Bspirit's Corrective Shape. Shape Builder is a UI script that wraps itself around Breitling's Corrective Shape script to add workflow tools that enable the quick and easy setup of generated shapes.

By doing my research, I was able to find a quick and solid solution that undoubtedly saved me many hours of work. In addition, I built a great friendship with Christian. If you have been in this industry for any length of time, you will immediately realize two important facts:

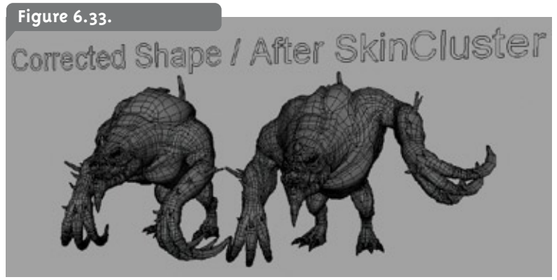
1. This world is really (I mean **really**) small. You will soon get to know some of the more vocal members of the online community. Those who give back to the community (through scripts, tutorials and forum help) quickly become recognizable and their skills grow. Those who do n't run the risk of neglecting their skills into obsolescence.
2. Colleagues are important. Building up a collection of contacts is instrumental in remaining competitive and current. You may meet at conferences, or in chat rooms, it does not matter. Having a good list of friends can give you guidance and encouragement when solving problems that come your way. This kind of support is especially fostered in a studio environment.

So, knowing how I came about the corrective shape script, let's discuss how it works. At the heart of the 'Bspirit Corrective Shape' script lies an algorithm, without which, none of this would be possible. The algorithm works by effectively subtracting the deformer-driven (ie skinCluster) vertex displacements on a mesh, leaving behind only the displacements that were modeled-in by hand. This creates a mesh with only the modeled corrections (and nothing else) 'baked' in. While Shape Builder was designed to hide the actual corrected shapes from the user, you can see what they look like by un-checking the 'Delete Corrective Shape' check box under the 'Generate Shape' button. You may notice that the actual blendshapes used look nothing like the shapes after they are applied FIGURE6.33. This is because corrected shapes are intended to be applied *after* other deformations. This is why they are called corrective shapes, they *correct* the problems left by other deformers (like skinClusters).

The method that Christian Breitling employs to subtract the deformed displacements from the modeling corrections involves some pretty complicated mathematics. Unfortunately, it is not just a matter of finding the world space displacement vector for each vertex. Rather, the algorithm must transform the modeled displacements from pose-space into world space. Coordinate space transformations are computed using matrix math. Do not be alarmed if you don't fully understand what

this means. It's nothing insanely difficult to comprehend or magical, but it does need some explanation.

Figure 6.33.



In fact, this algorithm will act as the perfect introduction to matrix math in general, and hopefully explain why matrices are so important to everything in 3d.

■ What is the Matrix?

Contrary to popular believe, a matrix is not actually a virtual reality created by evil robots for the purpose of enslaving mankind. Matrices are, like vectors, a type of data structure. If a vector is a 1 dimensional array of scalars, then a matrix can be described as a two dimensional array of scalars. Matrices are described in terms of rows and columns. Every matrix can be described as being of size R x C (pronounced 'r by c'), where 'R' is the number of rows and 'C' is the number of columns.

The study of matrices is deep and closely related to the branch of mathematics known as linear algebra. While it is far beyond the scope of this book to cover matrix operations (like addition and multiplication), I would like to impart to you the very basics of how matrices can be used in the hopes that you will decide to study them further.

Recall that a matrix is composed of rows and columns. When a matrix has the same number of rows as columns, it is said to be a *square* matrix. Square matrices are of particular importance in 3d math because they can describe a coordinate space transformation. A 3x3 (pronounced three by three) matrix can be used to describe what is known as the primitive transformations. This subset of transformations includes rotations, scaling, orthographic projection, reflection and shearing. If we add one more row and column to create a 4x4 matrix, we can describe what is known as an affine transformation. Affine transformations include translation.

When you toy with the transform values in the channel box, Maya is manipulating matrices to describe exactly what it is that you are doing to your object (like scaling, rotating and translating). When you manipulate the parent/child relationships of transform nodes (like in the hypergraph), Maya is doing all of the necessary coordinate space transformations to make your objects behave as expected.

A matrix can be constructed to describe any transformation on a point in 3d space. In the case of Maya's deformers, they are just C++ programs designed to transform a point in a particular fashion. That is all that a deformer really does. It builds a matrix that can transform the points in your mesh according to some pre-defined, desired behavior. The task of constructing the proper matrix to describe the desired deformation is the main problem in deformer design.

You might be wondering then, how exactly does a matrix transform a point (like a vertex or CV)? The answer is multiplication. A vector in 3d space (a) is multiplied by a square matrix (M) to produce a new point (b).

For a*M=b, M is said to have transformed a to b.

Multiplying a 3d vector by this 4x4 matrix will *mirror* your point across the x axis:

```
| -1 0 0 0 |
|  0 1 0 0 |
|  0 0 1 0 |
|  0 0 0 1 |
```

This matrix will *translate* your point by 4 units in X, 5 in Y and 6 in Z:

```
| 1 0 0 0 |
| 0 1 0 0 |
| 0 0 1 0 |
| 4 5 6 1 |
```

Any introductory 3d math text will describe the exact details for constructing a matrix for any of the most common transformations. It may be interesting to know that any number of transformation matrices can be concatenated into a single matrix that will describe an overall transformation. This is done using matrix multiplication. Even more interesting is that the order in which matrices are concatenated can greatly vary the number of scalar operations needed to compute the product. Optimizing the order of multiplications is known as the matrix chain problem.

The rules that describe matrix operations (like addition and multiplication), are actually quite straightforward. That being said, as anyone who as ever tried it using a pencil and paper will attest, it is quite mind numbing and tedious. Mind numbing and tedious you say? Well then my computer should be great at it! Indeed, computers are well suited to these kinds of operations.

MEL actually features a rarely used data type specially suited to matrices. The syntax used to declare a matrix looks a lot like an array:

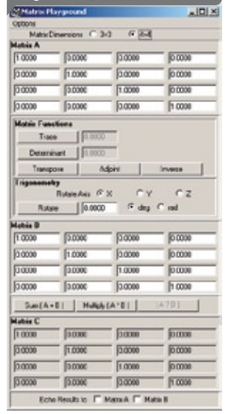
```
matrix $myFirstMatrix[3][3]; //Declare a variable of type matrix.
$myFirstMatrix[0][0] = 42.5; //Assign first row/column.
//Prints the first element of the most useless matrix ever.
print $myFirstMatrix[0][0];
```

The matrix data type has some caveats you should be aware of. Firstly, it only holds floats. String values will be converted to 0 at runtime. Secondly, MEL does not support the post-multiplication of a matrix by a vector to transform it. This unfortunate limitation can be overcome by creating procedures that do the operations explicitly. Thirdly, the matrix data type is not dynamic. You must specify the number of rows and columns when you declare the matrix and it will never change size during runtime.

```
//First Caveat
matrix $holdsOnlyFloats[3][3] = "hello"; //Not allowed.

//Second Caveat
matrix $myTransformMatrix[3][3];
vector $myPointIWantToTransform;
vector $newPoint = $myPointIWantToTransform * $mySecondMatrix;

//NOT ALLOWED

//Third Caveat
matrix $sizeNotSpecified[][];

// Error: Matrix declaration for "$sizeNotSpecified" requires
specification of two dimensions. //
```

I hope that you decide to study linear algebra further (if you haven't already). There are many great 3d math books available. Many programmers have developed C++ and MEL examples that utilize matrix math. You can find these scattered all over the Internet and they can be a great learning resource. Lastly, I want to mention a tool that I have found extremely handy for learning matrix transformations, matrixPlayground.mel by Brian Ewert (download from www.ewertb.com) FIGURE6.34. Matrix Playground is an excellent way to visualize exactly how matrices transform points. You can even load any Maya object to view it's transformation matrix, or affect any object with a matrix that you have constructed yourself. I think most artists are visual learners and tools like these can help a lot. Believe it or not, Maya is the perfect tool to learn matrix math with!

Figure 6.34.



For the MEL example at the end of this chapter, I will describe the matrix operations being performed, but it would be too long and unwieldy to describe every operation down to the scalar level. For this level of understanding, you will need either a 3d math text or to take an introductory linear algebra course.

■ Corrective Shapes Across the Body

You should now have a clear understanding of exactly what a corrective shape is and how they are generated. Perhaps what is more interesting than *how* they are made is the *why*. In exercise 6.1, we sculpted a corrective shape that fixed some deformation issues in the calve and thigh of our Dassak creature. While that was interesting and useful in it's own way, it was just one example of how to use corrective shapes throughout the body. For a more thorough understanding of where and why to use corrective shapes, you must first understand how different areas of the body stretch and compress (a notion similar to squash and stretch).

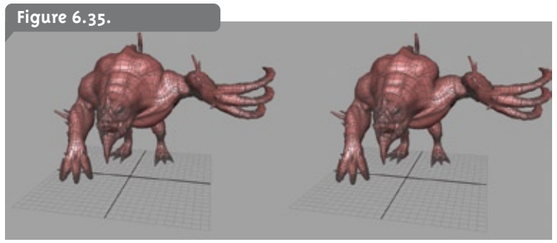
In general, smooth skinning creates excellent results in areas of the body under stretching tension (like the front of a knee). This is because, when skin is stretched, it does not necessarily change volume. Although stretching skin *can* sometimes cause a small *loss* of volume, it never causes bulging. For this reason, corrective shapes are rarely applied to areas under stretching tension.

Where smooth skinning falters is where it tries to compress the skin. There is no element in the smooth skinning algorithm that tries to account for the bulging effect of compressed flesh. This is why areas of the body under heavy compression are susceptible to poor looking deformations. Corrective shapes are most often used in these instances.

To help you find and fix the common problem areas throughout the body, let's take a look at some before and after pictures of Dassak.

The Shoulder FIGURE6.35: The shoulder joint has 3 axis of freedom, and as such, is best setup with a multi-axis driver. Set driven keys may be acceptable for some applications, but be aware that they have their limitations. The most common problem with shoulders is the loss of volume that occurs in the shoulder region when the arm is rotated into the 'da Vinci' pose. This can be minimized by modeling the arm at a 45 degree angle, but many riggers debate that this creates even more problems with joint orientations. The truth is that an arm can be properly setup in the 45 degree pose, it just requires some tuning of the joint orientations. The other argument is that arms modeled in the 'da Vinci' pose are difficult to deform properly because the armpit is under a lot of compression when the character is posed into a relaxed state (arms at the side). Depending on how your model was posed, you may need to create several corrective shapes to sculpt-out the loss of volume in this area. Also, don't forget to bulge-up the shoulder muscles when the arm is raised. You may also wish to create a fix for the 'hands up' pose where smooth skinning will usually create some glitches.

Figure 6.35.



The Hips FIGURE6.36: When I was sculpting deformations for Dassak's hip joint, I spent more of my time perfecting weights than actually correcting them with shapes. When the leg is pulled backwards (like an anticipation for a soccer kick), smooth skinning will push the bum cheek inwards. This area must be fixed in both quadrupeds and bipeds. Similarly, raising the leg forward (this action is actually only possible in really flexible creatures), will cause the stomach to cave in. This can be easily pushed out with another shape. When the creature's leg is kicked out to the side (like a ballerina), the loss of volume on the outside of the hips can be fixed as well. If you are using a multi-axis driver on the hip, you can sculpt a minor fix for the twisted hip joint as well.

The Knees FIGURE6.37: If you haven't already, check out exercise 6.1 for a full description of how calve and thigh interactions can be simulated with only one corrective shape. To get even more detailed, you can sculpt several different shapes for the calve and thigh to represent varying levels of compression. The outside of the knee is usually fine with smooth skinning alone.

Figure 6.36.

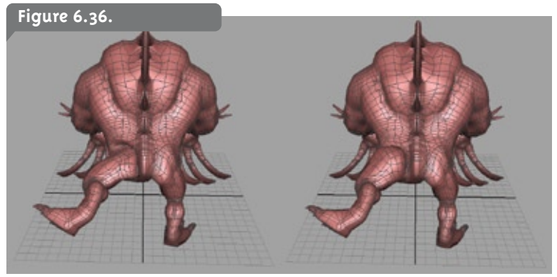
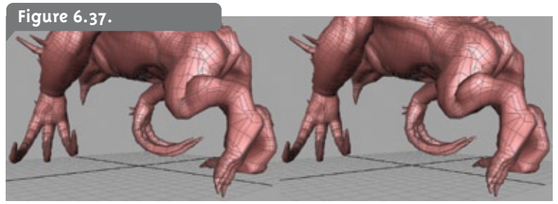
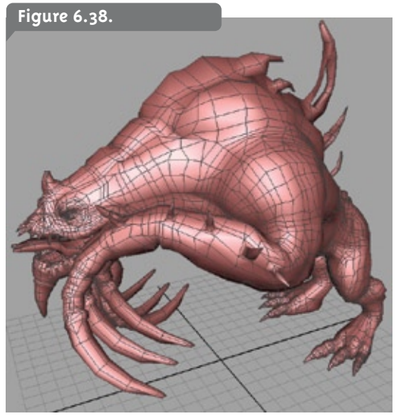


Figure 6.37.



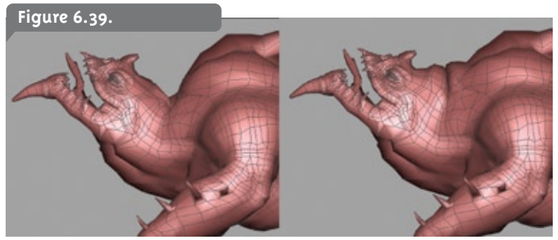
The Elbows FIGURE6.38: Like the knees, elbows are hinge joints that can only rotate on one axis. Shapes in this area should take care of the forearm-bicep collision problem. Try to bulge the bicep a bit and flex the triceps as well.

Figure 6.38.



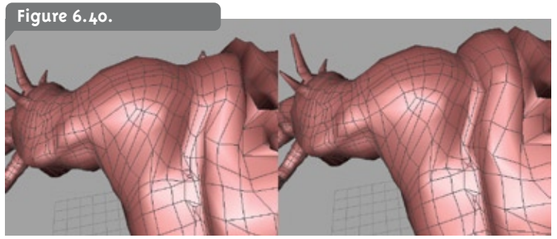
Head and Neck FIGURE6.39: While these areas are free to rotate in three axes, the range of motion is usually short enough that they do not need multi-axis drivers. When the head tilts backwards, try to sculpt-in some creasing and show the folds of fat bunching up. When the head looks side to side, you may notice the neck collapsing a bit. Sculpt this out and add a fold or two depending on how fat your character is.

Figure 6.39.



Clavicle FIGURE6.40: The clavicle is a tricky area. While it is technically not really capable of twisting, some animators like to twist it slightly to accentuate a pose. The obvious correction must be made when the shoulders are raised upwards. In this pose, the trapezius muscle is under extreme compression and should bulge outwards quite a bit. Real life reference can help you gauge how much to push this deformation. Rotating the shoulders forward and back may require some minor fixes as well if the mesh gets pinched.

Figure 6.40.



Wrists FIGURE6.41: The wrist joint does not twist. Beginner riggers always make this mistake. Just remember that twisting your hand starts at the forearm and is the product of the complex interaction between the radius and the ulna forearm skeleton. This can be simulated with an extra joint between the elbow and the wrist. That being said, there are other deformations to be considered in the wrist. When the wrist is pulled upwards, the skin at the top will bunch up and form many small wrinkles and creases. Try to sculpt these into a correction. The same thing happens when the wrist is pushed downwards.

Figure 6.41.

Fingers FIGURE6.42: For most creatures, fingers will need very few, if any, corrective shapes. If your creature has stubby sausage fingers, you will need to add bulging shapes for each knuckle. If the creature is intended for a video game or is an ancillary character (only seen in the distance), do not waste time sculpting corrections for each knuckle, smooth skinning should work fine.

Spine FIGURE6.43: When the creature bends forward above the waist, you should try to add some fat bunching and push the belly slightly (to simulate the internal organs squishing outwards). When the creature is posed, bending to the side, creases will form in the love handle region as fat and tissue bunch up and form large folds across the side. These corrective shapes can be difficult to sculpt because you must first correct the inevitable pinching before adding the bulging folds. Because the spine can twist as well, you may need to sculpt some minor pinching corrections to be driven by the twisting axis.

Figure 6.42.

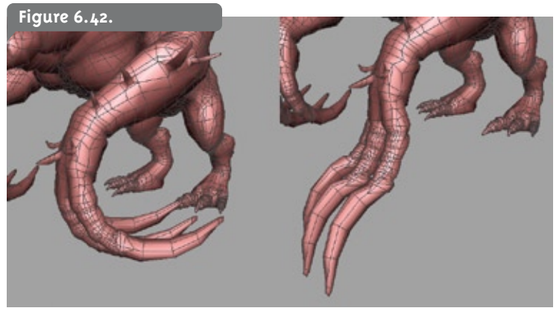
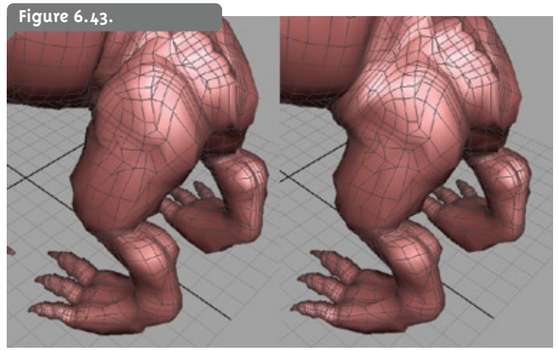


Figure 6.43.



These are the major areas of the body that require special attention. It is difficult to discuss, in extreme detail, the exact squash and stretch effects that occur throughout the entire body. This is largely due to the fact that these effects vary from creature to creature and between humans as well. Of course, real-life reference can provide the sort of details that lend a feeling of believability to your work. As with everything artistic though, this takes heaps of patience and practice. With time, you will become faster and more accurate at sculpting corrective shapes.

PSD really is the most artistic method of deforming a mesh I have ever had the pleasure of using. I hope you can appreciated it's simplicity and create some truly amazing creatures to wow your audiences with. Above all else, have fun with it!

■ Jiggly, Wiggly, Flabby, Fatty Flesh

To round-off our discussion of deformation techniques, we must talk about jiggle. Pose space deformation can provide the correct form and squash/stretch motion of the muscles and fat throughout the body, but this is not the only motion we must account for. Physical forces (like gravity and inertia), are always affecting bodies in motion. Character animations with fast movements tend to look very robotic if jiggle is not present.

Just like springs, every piece of flesh in your body is held together with a complex network of stretchy tendons and skin. These organic fasteners are not stiff. They have some flex and are capable of stretching. When a creature undergoes quick movements (like that of a run cycle), the bodies flesh is subjected to inertial forces that pull it back and forth. When the creature stops moving, these forces diminish and the jiggle effect gradually fades away. In physics, this is called dampening and it is caused by friction. Friction is created when the internal organs collide with the flesh and bones to absorb the kinetic energy. Friction is just a loss of energy, nothing else.

There are plenty of ways to simulate this effect in your creatures (mostly with the clever use of Maya's soft-body dynamics), but up until recently it has been very difficult to do so without using an added deformer or influence object. This was problematic because, as we already know, these types of deformations are not supported by game engines.

The technique presented in this chapter uses regular Maya joints to deform the mesh and simulate jiggle. Using nothing but regular Maya joints allows this jiggle effect to be exported to a game engine.

Well, that is all good and true, but surely you cannot expect the animators to hand animate 20 or so extra joints for each shot (or cycle)! So the problem then is how does one make these jiggle-joints dynamic? If you have read through chapter three, you have already been exposed to several different ways to make joint chains dynamic. While, in chapter two, this was done to animate antennae or floppy tails, we will now be using it for muscle and fat jiggle. Of course, the best dynamic chain technique in Maya is using the dynamic Hair curve (via spline IK) technique. If you have not read chapter three, please do so before continuing with the following exercise.

Exercise 6.2 Muscle and Fat Jiggle

In this exercise, we have been given a game cycle for the Dassak monster. If you open up exercise6.2_Start.mb, you will see that the animator has finished his walk cycle. Dassak is now ready to have his secondary jiggle simulated and baked in. To do this, we will be using cgTkDynChain.mel (included on the DVD). This script will allow us to create several small (one bone length), dynamic chains in key areas around Dassak's body. After adjusting the dynamic chains, we will run the simulation and allow the hair to affect his skin. When we are happy with the simulation, we will bake the animation into his joints and optimize the resulting animation curves. This will prepare him for export to a game engine.

1. Open exercise6.2_Start.mb. This scene file contains Dassak's finished walk cycle. If you playback the animation, you will see Dassak walk across the 36 frame cycle. This scene file contains Dassak's iconic representation rig. This was designed to give the animator an easier way of animating the joints. The keyframes themselves are actually on the curve based controls FIGURE6.44.

2. If you pop into wireframe mode and switch the rig layer on, you will see Dassak's underlying skeleton FIGURE6.45. While the majority of the skeleton may look familiar, you may wonder about the series of small two-joint chains that are attached to the major portions of his body. These small joint chains are the jiggle chains. These bones will be made dynamic to allow them to flop around with

his walk cycle.

Figure 6.44.

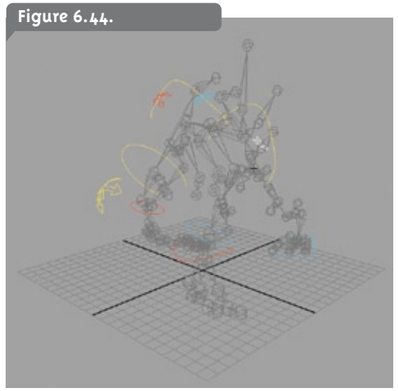
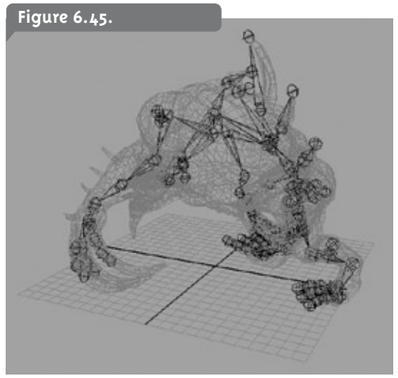


Figure 6.45.



3. These joints were added after all of the weighting and corrective shapes were added. In fact, we can even add jiggle joints after animation. The joint layout will need to be finalized before exporting it to a game engine, but we can add joints during the test phase. It may help to have some animation to test the jiggle joint layout. A walk cycle is perfect for this because it involves all areas of the body. While the majority of Dassak's body has already been setup for jiggle, it has been decided that his flanks are quite large and bulky. So much so that they could definitely benefit from some muscle jiggle.

4. To add a joint to Dassak's flank region, we need to get him into his default pose. Because his joints are currently being controlled by constraints and inverse kinematics, we must disable these before we can reach the bind pose. Click on `Modify > Evaluate Nodes` and then un-check 'IK Solvers' and 'Constraints'. Now select one of Dassak's joints and select `Skin > Go To Bind Pose`. Now Dassak is ready to have some joints added FIGURE6.46 .

5. Drop into a front viewport and create a two-joint chain in this position FIGURE6.47. The length of the chain will determine the radius of the jiggle motion. Position the chain into Dassak's flank region. To mirror this to the other side, duplicate the chain, group it to itself and type a value of -1 into the scale X of the duplicate group.

Figure 6.46.

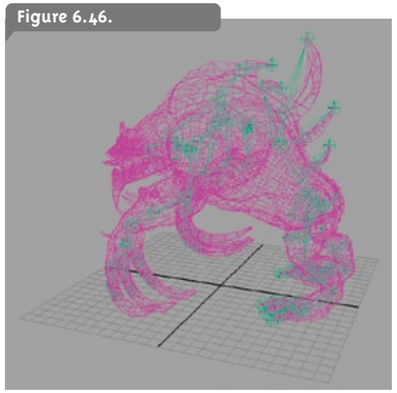
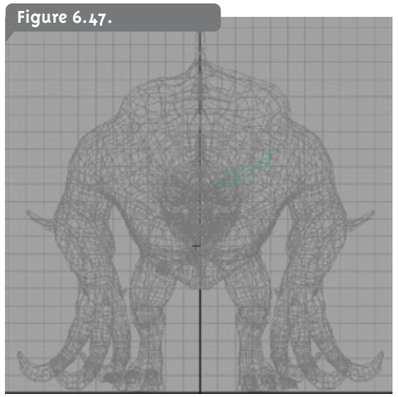
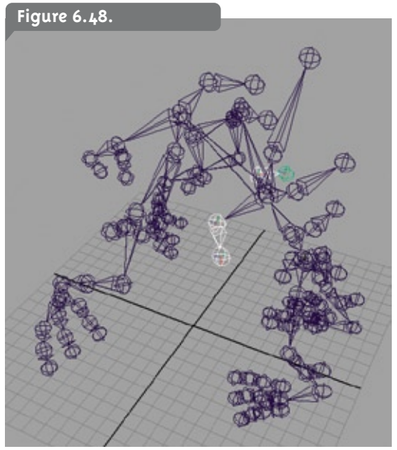


Figure 6.47.



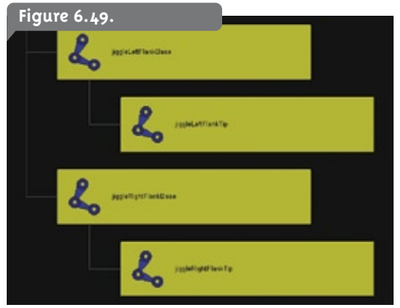
6. Un-group the duplicate chain. Now parent these chains to the spineA joint. This will cause the flank region to inherit secondary motion from the motion of the spineA joint FIGURE6.48.

Figure 6.48.



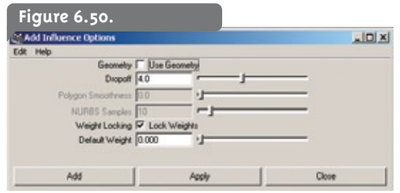
7. To stick with the naming convention used with the rest of Dassak, rename these joints. The left chain joint should be called jiggleLeftFlankBase and jiggleLeftFlankTip. Name the right joints accordingly FIGURE6.49 .

Figure 6.49.



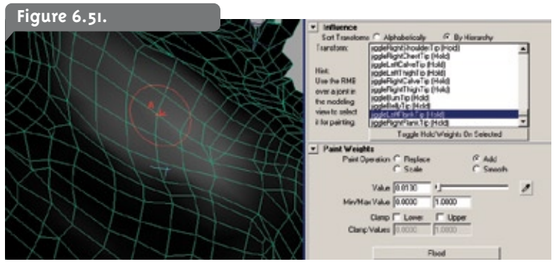
8. Open exercise6.2_MiddleA.mb to see the progress this far. These joints are now placed, parented and named properly but they still do not affect the skin. To reduce the number of skinned joints, we needn't add the base joint into the skinCluster. The base joints act as anchors for the tips to rotate from. Select Dassak's mesh, then shift select the left tip joint and choose `Skin > Edit Smooth Skin > Add Influence`. Go into the options box and uncheck 'Use Geometry'. Check the 'Lock Weights' box and hit apply FIGURE6.50. Do the same for the right tip joint.

Figure 6.50.



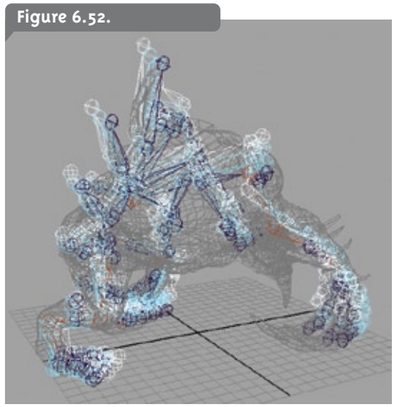
9. The newly added joints are now added to the skinCluster. Because we added them with the 'Lock Weights' option, they do not have any default weighting. This allows us to use the artisan brush to add some custom weight for these joints. Select the mesh and choose `Skin > Edit Smooth Skin > Paint Skin Weights` Tool. In the tool options, scroll down to the bottom of the list to see the newly added joint. Select the jiggleLeftFlankTip joint and use the 'add' operation with a value of '0.01' to paint influence FIGURE6.51. Use small, quick, brush strokes to paint a subtle influence across Dassak's back. To avoid messing up the skin weighting, you should avoid mirroring the weighting on this joint. Try to gradate the effect into the surrounding regions. If you need to see exactly how the joint is affecting the mesh, select the tip joint in the perspective viewport and move it around. You should see a very soft effect on the skin with no harsh edges. This will give the best jiggle behavior. When you are done weighting the new flank joints, continue on with the jiggle setup.

Figure 6.51.



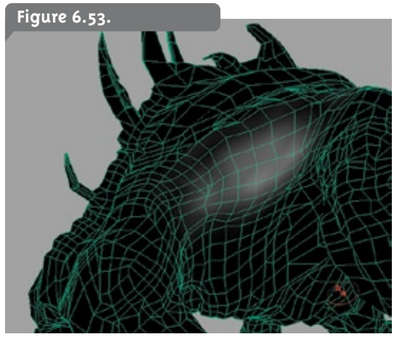
10. Before we setup the dynamic joint chains, let's re-enable all of the IK solvers and constraints. Choose `Modify > Evaluate Nodes > Evaluate All`. If you scrub in the timeline, the animation should be back again FIGURE6.52 .

Figure 6.52.



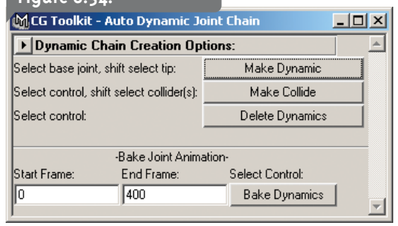
11. Open exercise6.2_MiddleB.mb to see how the flank joints behave after being added to the skinCluster and weighted to the surrounding areas. If you open the paint weights tool, you can see the weighting that I used for these joints FIGURE6.53 .

Figure 6.53.



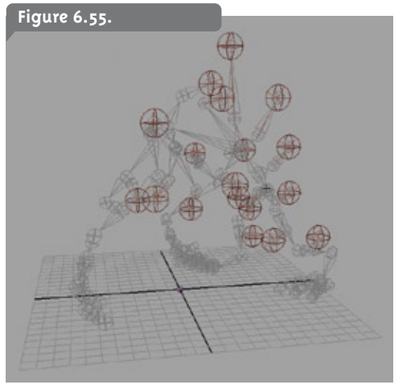
12. Now we are ready to start the simulation. To make these chains dynamic, we will be using the cgTkDynChain.mel script. If you haven't already, copy this script to your maya/scripts directory and open it by typing 'cgTkDynChain;' in the command line FIGURE6.54. This is the same script from chapter 3.

Figure 6.54.



13. Go through Dassak's body and create dynamic chains on each of his jiggle joint chains. Select the base joint, shift select the tip and hit the 'Make Dynamic' button in the script window. There are jiggle chains located all over Dassak's body. Create a dynamic chain for the left/right flanks, left/right calves, left/right thighs, left/right shoulder, left/right chest, the bum, belly, neck and along the spikes on his back. This will result in the creation of eighteen different jiggle points. For your own creatures, I would recommend adding as many jiggle joints as your production can handle. The more points the better. Eighteen, however, provides a very nice effect without slowing things down too much FIGURE6.55.

Figure 6.55.



14. Open exercise6.2_MiddleC.mb to see Dassak with all of his dynamic chains. Because we are simulating a cycle (and not a linear animation), we must extend the animation to allow the dynamics to settle into a rhythmic pattern. Select all of the animation curves (there are 12 in total) and open the graph editor. In the graph editor, choose `Curves > Post Infinity > Cycle` FIGURE6.56. This will cause Dassak's animation to cycle for infinity. Now extend the frame range to play for 1000 frames and set the playback to 'Play Every Frame'. This will provide enough room to allow the dynamics to settle into a rhythm.

15. Open exercise6.2_MiddleD.mb to see Dassak with all of his dynamic chains and the extended animation. If you now hit play, you will see his dynamic chains wobble around with his animation. While some parts of the body are already looking much better (the spikes should be moving quite convincingly), there are some trouble spots. The dynamic chains are way too floppy in some of the fast moving areas. This is causing some crashing as the chains push the mesh around and pinch it FIGURE6.57. To fix this, we need to adjust the stiffness of the chains.

Figure 6.56.

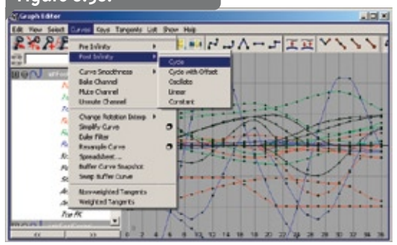
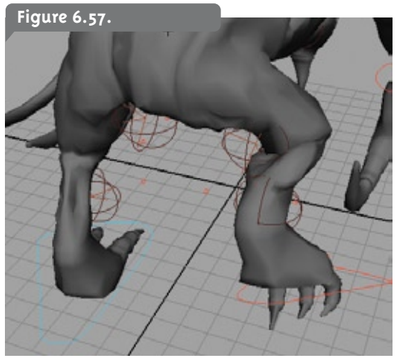


Figure 6.57.



16. Set the following stiffness values:
Calves = 0.08
Thighs = 0.03
Shoulders = 0.03
Pectorals = 0.06
Neck = 0.03
SpikeA and SpikeB = 0.08
Bum and Belly = 0.03

17. Open exercise6.2_MiddleE.mb to see Dassak with all of his dynamic chains tweaked. Play the scene to watch the chains settle into a rhythm. These chains are ready to have their dynamics baked into keyframes on the joints FIGURE6.58. Do this by selecting each chain's spherical controller and clicking on the 'Bake Dynamics' button. Specify a frame range of 0 to 600. This will provide enough time to allow the dynamics to form a tight rhythm. After baking each chain, select the sphere controller and hit 'Delete Dynamics'. This will clean up the hypergraph leaving only the joints. You may find it helpful to turn off Dassak's mesh layer while baking. This will speed up the baking considerably.

18. Open exercise6.3_Finished.mb to see Dassak's animation with the baked in dynamics. Notice that playback is quite fast now that the dynamics do not have to be calculated. To squeeze more speed out of this animation, for export to a game engine, you may consider simplifying the animation curves on the dynamic joints. This can provide a further speed bonus. This is done in the graph editor menu, `Curves > Simplify Curve` FIGURE6.59.

Figure 6.58.

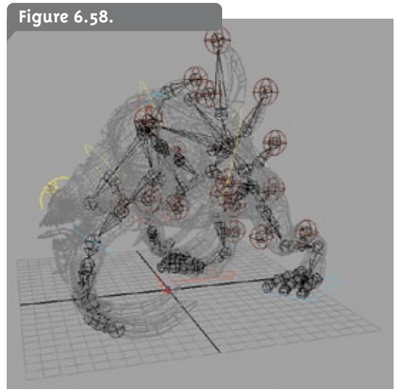
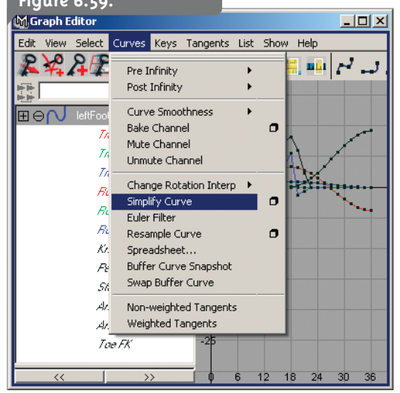


Figure 6.59.



This same technique can be used in film and television creatures as well. In fact, this technique has given me far better results, far quicker than any other jiggle technique I have ever tried. Because it is done with regular joints, the overhead is negligible and the affect on the skin is easily adjusted using the artisan weights tool. It really is a very simple workflow, with fully controllable results and minimal overhead. I hope your next production can benefit from these techniques.

MEL Scripting: The creation of cgTkShapeBuilder.mel from A to Z

The Shaper Builder script described here was created by both myself (Kiaran Ritchie) and a German programmer/TD/nice guy, by the name of Christian Breitling. The core procedure, the one that generates corrective shapes, was created by C. Breitling and adapted by myself (with Christians permission) to form Shape Builder. The script is a sort of blendshape based PSD solution, designed to give video game artists a PSD-esque rigging experience without having to use a custom deformer (that would render the rig unusable in a game engine).

After discovering Breitling's Corrective Shape script (as described earlier in this chapter), I set about to design exactly how I wanted Shape Builder to work. The main reason for adapting Breitlings script was to give it a sort of home to live it. While the Corrective Shape script, on it's own, was an excellent tool, I felt it could benefit from a more user friendly front-end.

I wanted this front-end interface to allow the user to not only create corrective shapes, but hook them up as well. All this in a nice little floating window. To start with, I researched how other PSD front-ends looked. My first clue came from a 3ds max demo video where the 'Sculpt Morpher' (basically the Sculpt Morpher is 3ds max's version of PSD) was shown off. In this video, the setup artist used a button to quickly access a graph editor to modify the interpolation of the one of the shapes. A light bulb went on. I thought, "Hey! Maya has a graph editor, maybe I can do the same thing". In addition to the demo video, I read a couple SIGGRAPH papers, asked some friends (who use other software) and poked around the MEL script section at highend3d.com.

And so I went snooping around stealing ideas from everything I could get my hands on until I finally decided to sit down and flesh it out on paper. When you create your own scripts, always do some snooping of your own. You never know what sort of ideas you might find. Thanks to my research, I was left with plenty of ideas for both the interface and the execution of my blendshape based PSD solution.

The interface is designed with workflow in mind. Notice that the 'Create Shapes' tab is located to the left of the 'Setup Shapes' tab FIGURE6.60. This is done on purpose because the general workflow for using this script involves first creating a shape and then hooking it up. Going from left to right makes it feel like a natural progression. In fact, I could have mashed the whole thing into one single tab and added a scroll bar, but that would have been a bad idea. It is annoying to have to constantly scroll up and down as opposed to simply clicking on a tab. Of course, I could have also just made the window really big in order to accommodate all of the interface elements. This would have been the worst idea because of the amount of screen real estate it would take up. In fact, keeping the window small was a major priority because workflow scripts, like this one, are kept open for the duration of their use.

Figure 6.60.

Interface design is a weird art. I don't pretend to be an expert at it, but common sense and good judgment can help a lot. With these considerations in mind, you will create scripts that are more accessible, easier to understand and thus faster to adopt and become useful.

So let's get started then. And the first thing we need to discuss? You guessed it, the interface!

The Shape Builder Interface

As with all of the scripts presented in this book, Shape Builder was designed starting with the interface. This allows you to start working with the program as you are building it. This often leads to better ideas about exactly how to implement various features as well as providing a quick way to test the code as you write it.

The entire interface is contained within a global procedure called 'cgTkShaperBuilder'. This is the procedure that must be calledwhen the

user wishes to start using the script.

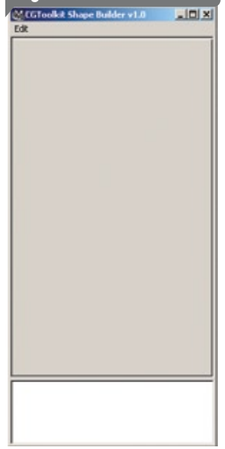
```
global proc cgTkShapeBuilder ()
{
//Check for multiple similar windows.
if (`window -q -ex shapeBuilderWindow`)
deleteUI shapeBuilderWindow;

//Build window.
window -w 280 -h 580
-title "CGToolkit Shape Builder v1.0" shapeBuilderWindow;
```

This code should be familiar by now. The if statement checks for the existence of the shaperBuilderWindow and deletes it, if necessary, to prevent multiple copies from being open at once. Then we continue on with the window command and specify both the initial dimensions (280 x 580) and the title, 'CGToolkit Shaper Builder v1.0'.

```
//Build 'Edit' menu.
string $menuBarLayout = `menuBarLayout`;
menu -label "Edit";
        menuItem -label "Load Base Mesh"
                 -command "loadBaseMesh;";
        menuItem -label "Load Active Blendshape Node"
                 -command "loadBlendshapeNode;";
        menuItem -label "Close"
                 -command "deleteUI shapeBuilderWindow";
```

At the top of the interface hierarchy, we must add our menu bar. For Shaper Builder, I only needed one menu called 'Edit'. This menu is how the user will load the base mesh (the mesh that they are creating shapes for) and the active blendshape node (the blendshape node that they want the newly created shapes to be added to). The menu bar layout is simutaneosly created and stored into the $menuBarLayout string. To add a menu, simply use the 'menu' command followed by a label (-label "Edit"). The menu items are added with the 'menuItem' command. These are quite straightforward and self explanatory. The procedures called by the first two menu commands are covered later.

Figure 6.61.




```
//Create Pane Layout
paneLayout -configuration "horizontal2" -paneSize 1 10083;

        //Top Pane (The Tabs)
        tabLayout;
           //Create Shape Tab
           columnLayout -adjustableColumn true "Create Shapes:";
```

Continuing down the interface hierarchy, I first added a pane layout. This layout's sole purpose is to create a horizontal divide with which I can place the tabs in the top, and the currently loaded element information in the bottom FIGURE6.61. This is to ensure that the user always knows which base mesh and blendshape node are currently loaded, regardless of which tab they are in. This is pretty fundamental information and should be easy to see at a glance. The script then goes on to create a tab layout. Tab layout's are unique in that they require their first child to be another layout. The name of the first child becomes the name of the tab itself. In this case, I made a column layout and named it "Create Shapes". This creates a tab with the name "Create Shapes". The tab layout will create a new tab for every direct descendant layout. When I am finished with the column layout, I will need to use a 'setParent..;' command to create another tab (the 'Setup Shapes' tab). Continuing with the 'Create Shapes' tab, I need to created a few buttons and some text to describe how to use them:

```
separator -h 10;
text -align "center" -label "- Create Corrected Blendshapes -";
separator -h 10;
text " ";
text -align "left" -label "Step #1 :";
text -align "center"
-label "Pose a joint. Select base mesh,shift select the joint.";
button -label "Sculpt Changes on Duplicate"
        -command "createSculptDuplicate;";
separator -h 10;
text " ";
text -align "left" -label "Step #2 :";
text -align "center"
     -label "Once sculpting is finished, select duplicate.";
button -label "Generate Blendshape."
       -command "generateBlendshape;";
checkBox -label "Delete Corrective Shape?" deleteShapeCheckBox;
separator -h 10;
setParent..;
```

You will notice in the finished interface, that each button is separated by a single gray line. This is called a 'separator' and they can be used to provide a clear distinction between different parts of your interface. This tab is quite straightforward. No tricks going on here. You may wonder about the use of the 'text " "' commands. Why would I want to print out nothing? These commands are simply placeholders to fill up an entire row in the column layout. Think of them as setting the cursor down a line (like hitting the `enter` key in a word processor). They are used to simply make white space.

The two buttons in this tab call the createSculptDuplicate and generateBlendshape procedures. These are covered in detail later on. For now, just recognize that this is where they are called from. You may also be wondering about the deleteShapeCheckBox. This is used to specify whether the user wishes to have the corrective shape deleted after it has been created and added into the blendshape node. Users may wish to mirror this shape, in which case automatically deleting it would be detrimental. Lastly, the 'setParent..;' command is used to set the hierarchy back to the tab layout. This allows us to create the next tab:

```
//Setup Shape Tab
columnLayout -adjustableColumn true "Setup Shapes:";
        separator -h 10;
        text -align "center"
             -label "- Connect Blendshapes To Joints -";
        separator -h 10;
        text " ";
        text "Blendshape List:";
        textScrollList  -numberOfRows 8
        -selectCommand ("textField -edit -text ...
        (`textScrollList -q -selectItem ...
        blendShapeTextScrollList`) drivenShapeTextField") ...
                                    blendShapeTextScrollList;
        button -label "Refresh Blendshape List."
                -command "refreshBlendshapeList;";
        separator -h 10;
        text -label "Driven Shape: ";
        textField -editable false "drivenShapeTextField";
        button -label "Load Driver Joint:"
                -command "loadDriverJoint";
        textField -editable false "driverJointTextField";
        separator -h 10;
        radioButtonGrp    -columnAlign 1 "left"
                          -columnWidth4 60 60 60 60
                          -numberOfRadioButtons 3
                          -label "Joint Axis:"
                          -labelArray3 "X" "Y" "Z"
                          -select 2
                          "jointAxisRadioButtons";
        floatSliderGrp    -columnAlign 1 "left"
                          -columnWidth3 65 40 60
                          -label "Joint Angle:"
                          -field true
                          -minValue -180.0
                          -maxValue 180.0
                          -value 0
                          -dragCommand "jointSliderCommand;"
                          -changeCommand "jointSliderCommand;"
                          "jointSlider";
        floatSliderGrp    -columnAlign 1 "left"
                          -columnWidth3 65 40 60
                          -label "Shape %:"
                          -field true
                          -minValue 0.0
                          -maxValue 100.0
                          -value 0
                          -dragCommand "shapeSliderCommand;"
                          shapePercentageSlider;
        button -label "Set Key" -command "setShapeKey;";
        button -label "Edit Curve" -command "editCurve;";
        separator -h 10;
        setParent..;
setParent..;
```

This tab is a little bit more complicated. Like the first one, it is all contained within a single column layout. This layout's name is then used as the name of the tab itself. The first actual interactive element is a text scroll list. These are very handy interface elements that can be used to hold a list of items. If the list is too long to be contained within the element, a small scrollbar is automatically created. In this case, I created one to hold all of the blendshapes in the currently loaded blendshape node. Populating this list is done with a completely separate procedure. At this point in the script, we are only concerned with creating it and specifying it's size (8 rows).

In order to actually make the text scroll list usable, we can connect a command to it that is executed whenever the user clicks on one of the items in the list. In this case, we want the user to be able to quickly load a blendshape from the list to be setup with set driven keys. By attaching this flag, '-selectCommand ("textField -edit -text (`textScrollList -q -selectItem blendShapeTextScrollList`) drivenShapeTextField")', we can have the program query the currently selected item, and load this into the drivenShapeTextField. This text field displays the current driven shape and is actually created in just a few lines after the scroll list.

The button right below the scroll list calls a procedure by the name of 'refreshBlendshapeList', this procedure is declared later to populate the list by querying the contents of the currently loaded blendshape node. While this list is updated automatically, the user may wish to add blendshapes into the node on their own. In this case, the list must be updated manually to reflect any newly added shapes.

The next two elements are text fields. These contain the current driver joint and the currently loaded blendshape (whichever one the user selected from the list). The only thing of interest here is that I used the '-editable false' flag to specify that the fields should not be editable. This is to ensure that the user does not accidentally edit the fields which are designed to house only objects that both exist and are of the correct type. To load the driver joint, a button calls a procedure called 'loadDriverJoint'. This procedure simply checks the current selection to ensure it is a joint, and then loads the name of the currently selected joint into the driver text field. This procedure is covered in detail later on.

The next element is a radio button group. This gives the user the option of selecting exactly which axis should drive the blendshape. Of particular interest here is the flag '-columnWidth4 60 60 60 60' which specifies the width of each radio button element. Without explicit control over the width, this group of radio buttons would be too wide to fit into our skinny little window (using default creation widths).

After specifying the driving joint axis, the user must specify both the value of the driver joint angle and the driven blendshape. These values are controlled with the use of two float sliders. Of particular interest are the two flags attached to each of these float sliders. The '-dragCommand' and '-changeCommand' flags allow you to specify a command that will get executed when the user either drags the slider bar or manually types in a new value. In this case, I setup these flags to call some procedures (jointSliderCommand and shapeSliderCommand) to affect the attributes in real time. These procedures query the current value of the slider and then use the setAttr command to set the joint or blendshape to the current value of the slider. The result of which is that the sliders affect the joint angle and blendshape weight as the user drags them. This makes for a very quick setup as the user needn't leave the interface to see how the blendshape and joint angles are interacting at the current state.

Lastly, there are two buttons in this tab that call the 'setShapeKey' and

'editCurve' procedures. When the user presses the 'Set Key' button, the 'setShapeKey' procedure will set a driven key using the values that are currently loaded into the sliders on the objects that are currently loaded into the text fields. The 'Edit Curve' button calls a procedure that simply brings up the graph editor with the current blendshape node automatically loaded. These procedures are covered in detail later in this chapter.

After all of the buttons, sliders and scroll lists are created, the 'Setup Shapes' tab is finished. To add the final interface elements we need to set the hierarchy back two spaces so that we can then add elements into the second pane. Recall that the menu bar layout has a child pane layout of which the tab layout is the first child. Adding two 'setParent..;' commands returns the hierarchy to the pane layout level. Now we can add the two text fields that display the currently loaded base mesh and blendshape node.

```
//Bottom Pane (The Information)
columnLayout -adjustableColumn true;
        text -align "left" -label "Currently Using Base Mesh:";
        textField -editable false currentBaseMeshTextField;
        text -align "left" "Currently Using Blendshape Node:";
        textField -editable false currentBlendshapeNodeTextField;
setParent..;
```

This is the entire code block for the bottom pane. Notice that both of these text fields have their editable states set to 'false'. This is because they are only meant to act as informational displays. These two text fields are edited whenever the user loaded a base mesh or a blendshape node. Because they must be explicitly edited during runtime, they are given unique names with which they can be identified.

The interface procedure is now finished. The only thing left to do is show the window and end the procedure declaration with the closing curly brace:

```
//Show the CGToolkit Shape Builder window.
showWindow shapeBuilderWindow;
}
```

■ The Other Procedures

This script is perhaps a bit more complicated than anything we have studied thus far because of it's sheer number of procedures. While it may be a larger and more ambitious coding endeavor, the actual procedures themselves are quite simple (with the exception of the corrective shape procedure).

In total, there are eleven other procedures, all of which are directly related to a particular button on the interface.

loadBlendshapeNode;
loadBaseMesh;
createSculptDuplicate;
refreshBlendshapeList;
loadDriverJoint;
jointSliderCommand;
shapeSliderCommand;
setShapeKey;
editCurve;
generateBlendshape;
BSpiritCorrectiveShape;

There are no 'floating' procedures or code blocks that are called from within other procedures. This is not necessarily a good or a bad thing, it's just the way this script is designed. So, to study the way all this was put together, let's take a look at each button and it's corresponding procedure.

Recall that this script, while fundamentally quite simple, may look more complicated because of the included error checking. While most of the commands and techniques used should be quite familiar, the error checking can cause them to look more complex. This is why it was avoided in the earlier chapters. But for this example, I will be explaining the entire, finished, production ready script. Complete with full error checking. This is done to give you a better understanding of both the types of errors commonly encountered in scripts like this, as well as to show you how to catch them.

■ The loadBlendshapeNode Procedure

This procedure is called from the 'Edit > Load Active Blendshape Node' menu item. When called, it will take the currently selected blendshape node (from the channel box) and load it into the blendshape node text field. This allows other parts of the script to know exactly which blendshape node is being worked on.

```
global proc loadBlendshapeNode()
{
 string $currentSelXform[] = `ls -sl`;

 if ($currentSelXform[2] != "")
 {
  warning "Please select a blendshape node from the channel box.\n";
 } else
 {
  if (`objectType $currentSelXform[0]` != "blendShape")
 {
   warning "Please select a blendShape node fromthe channel box.\n";
  } else
  {
   textField -edit
             -tx $currentSelXform[0] currentBlendshapeNodeTextField;

   print ($currentSelXform[0] ...
                         + " was loaded as the blendShape node.\n");

   refreshBlendshapeList;

   if ($currentSelXform[1] != `textField -q ...
                                      -tx currentBaseMeshTextField`)
   {
     warning "The newly loaded blendshape node does not belong ...
     to the current base mesh. Please load the proper base mesh.\n";
   }

  }

 }

}
```

At first glance, this procedure may look complicated because of the depth of nested if statements. All of this is because of the error checking. In fact, all of the procedures in this chapter will look like this.

The procedure starts by creating a local string variable called $currentSelXform. This is meant to read like 'Currently Selected Transform Nodes'. You should be familiar with the 'ls -sl' command. It simply returns a list of all of the objects in the scene that are currently selected.

The first if statement checks to see if the user has selected more than two objects. To avoid confusion about exactly what the script needs (just one blendshape node object) the script prevents this and makes the user select only two objects (the mesh and it's blendshape node in the channel box).

If the user only has two selected objects, the program flows to the next if statement. This 'objectType' command might be new to you. It really is the most important command for error checking. Using 'objectType' you can check to see what type of object is currently selected. In this case, we need an object of type 'blendShape'. The full list of various object types can be found in the MEL command reference. If the user has only two objects selected, and the first object is of type 'blendShape' then the program will flow to the next code block.

To actually load the blendshape node, the script uses the '-edit' flag to change the text value of the current blendshape node's text field. This text field was explicitly named during it's creation so that we could edit it in this procedure. If the text field had not been named, it would have been impossible to refer to it to edit it.

After loading the name of the node, we print some feedback to the user and call the refreshBlendshapeList procedure. The last if statement checks to see if the blendshape node that was just loaded actually belongs to the currently loaded base mesh. If it does not, a warning is output to the feedback line.

■ The loadBaseMesh Procedure

This procedure is very similar to the previous one.

```
global proc loadBaseMesh()
{

string $currentSelXform[] = `ls -sl`;
string $currentSelShape[] = `listRelatives ...
                                    -shapes $currentSelXform[0]`;

 if ($currentSelXform[1] != "")
 {
  warning "No base mesh loaded. Please select only ...
                                        1 polygonal mesh to load.\n";
 } else
 {
  if (`objectType $currentSelShape[0]` != "mesh")
 {
  warning "Base mesh must be a polygonal mesh. ...
                    NURBS and SUBD surfaces are not supported.\n";
} else
{

  textField -edit -tx $currentSelXform[0] currentBaseMeshTextField;

    print ($currentSelXform[0] + " was loaded as the base mesh.\n");

   }
  }
}
```

The first string variable is nothing new. The second one, however, is initialized to the returned value of the 'listRelatives' command. This command, with the '-shape' flag attached, will return the name of the shape node connected to the specified transform node. We need the shape node in order to check for the correct object type later in the procedure. In previous examples from other chapters, I used pick walking to find a related shape node. The 'listRelatives' command accomplishes the same thing.

The first if statement checks the number of selected items to ensure that the user only has one object selected. The second safe guard prevents the users from loading an object that is not of type 'mesh'. Finally, if all the conditions are met, the currently selected mesh is loaded and a nice message is printed to the feedback line.

■ The createSculptDuplicate Procedure

```
global proc createSculptDuplicate()
{
 string $currentSelXform[] = `ls -sl`;

if (`textField -q -tx currentBaseMeshTextField` == "")
{
   warning "Duplicate not created. Please load a base mesh ...
            before creating a duplicate. Edit>Load Base Mesh.\n";
} else
if ($currentSelXform[0] != `textField -q ...
                                    -tx currentBaseMeshTextField`)
{
  warning "Duplicate not created. Please select the currently ...
   loaded base mesh before trying to create a sculpt duplicate.\n";
} else
if ( ($currentSelXform[1] == "") || (`objectType ...
                                    $currentSelXform[1]` != "joint") )
{
   warning "Duplicate not created. Please select the base mesh, ...
                            then shift select the posed joint.\n";
 } else
{
  //Duplicate the mesh.
  string $nameOfDuplicate[] = `duplicate -rr $currentSelXform[0]`;

  //Create a list of locked attributes.
  string $UnlockAttributes[] =
          {".tx",".ty",".tz",".rx",".ry",".rz",".sx",".sy",".sz"};

//Unlock the transform attributes.
```

```
for ($i = 0; $i < 9; $i++)
{

   setAttr -lock 0 ($nameOfDuplicate[0] + $UnlockAttributes[$i]);

}


//Find the bounding box coordinates.
float $boundingBoxInfo[] = `polyEvaluate -boundingBox ...
                                          $currentSelXform[0]`;

//Subtract the Xmax from Xmin.
float $xtranslate = ($boundingBoxInfo[1] - $boundingBoxInfo[0]);

//Move the duplicate mesh the distance of the bounding box.
move -r $xtranslate 0 0 $nameOfDuplicate[0];

//Rename the new mesh.
string $newName = `rename $nameOfDuplicate[0] ...
                          ($nameOfDuplicate[0] + "blendShape1")`;

//Add custom attribute with joint rotation angles.
addAttr -ln xRotation -at double $newName;

addAttr -ln yRotation -at double $newName;

addAttr -ln zRotation -at double $newName;

addAttr -ln nameOfJoint -dt "string" $newName;

//Set custom attributes to house joint information.
setAttr ($newName + ".xRotation") ...
                          `getAttr ($currentSelXform[1]+ ".rx")`;

setAttr ($newName + ".yRotation") ...
                          `getAttr ($currentSelXform[1]+ ".ry")`;

setAttr ($newName + ".zRotation") ...
                          `getAttr ($currentSelXform[1]+ ".rz")`;

setAttr -type "string" ($newName + ".nameOfJoint")



$currentSelXform[1];


}

}
```

This procedure is called when the 'Sculpt Changes on Duplicate' button is pressed. This creates a duplicate of the currently loaded base mesh and moves the duplicate in the positive X direction based on the size of it's bounding box. The object is moved to the side to make it easy to start sculpting changes right away.

The first if statement checks to see if a base mesh is loaded. It nothing is loaded, the script returns an error. The second if statement checks to make sure that the user has selected a mesh that matches the currently loaded base mesh. This is to ensure that he/she creates a duplicate of the correct mesh (and not something that was not intended). The third if statement checks to make sure that the base mesh is indeed selected, and that the user has also selected a joint. The driver joint must be selected so that the current rotation values can be recorded into the duplicate. This helps the script reconstruct the desired pose before creating a corrected shape, as well as providing the setup artist with a convenient reminder of exactly which pose the shape was duplicated in. This can be a great help if you forget to write down the angle at which the shape was supposed to be blended-in at.

If all of the conditions are met, the mesh is duplicated and each transform channel is unlocked inside of the little for loop (remember that bound meshes have their transforms locked). After unlocking the transforms, we can move the mesh to the side to prepare it for modeling. The amount that the mesh is translated is calculated as the distance from one end of the bounding box to the other. This difference is added to the 'translate.X' of the newly created mesh.

After renaming the mesh, the pose with which it was created at is read-in from the selected joint and recorded into custom-added attributes. These attributes are needed to be able to reconstruct the exact pose so that the corrective shape procedure knows exactly what to deformations need to be subtracted. Without this recorded pose, the user could accidentally repose the driver joint after creating the duplicate causing the corrected shape to be constructed with extra garbage displacement on the vertices.

■ The refreshBlendshapeList Procedure

This procedure is called from the 'Refresh Blendshape List' button. When called, it queries the currently loaded blendshape node to find all of the blendshapes and load them into the text scroll list on the interface. This procedure is called from several other procedures as well. The list must be updated whenever the blendshape node changes.

```
global proc refreshBlendshapeList ()
{

  if (`textField -q -tx currentBlendshapeNodeTextField` == "")
  {

   warning "Blendshape list was not updated. Please load a ...
         blendshape node first. Edit>Load Active BlendshapeNode. \n";

  } else
  {

   //First we must remove all of the items in the list.
   textScrollList -edit -removeAll blendShapeTextScrollList;
   //Now we find the name of the currently loaded blendshape node.
   string $currentBlendshapeNode = `textField -q ...
                                  -tx currentBlendshapeNodeTextField`;

   //Using the listAttr command, we can get a list of all
   //the shapes in the node.
   string $listOfShapesInNode[] = `listAttr -k -m ...
                                             $currentBlendshapeNode`;

   //We initialize this string before adding to it in the loop.
   string $addShapesCommand = "textScrollList -edit ";//-append "one"

   //This loop adds -append "shapeName"' for each shape in the node.
   int $i = 1;

   while ($listOfShapesInNode[$i] != "")
   {

     $addShapesCommand = ($addShapesCommand + "-append \"" + ...
                                        $listOfShapesInNode[$i] + "\" ");

     $i++;

   }

   //Add the name of the text scroll layout.
   $addShapesCommand=$addShapesCommand + "blendShapeTextScrollList;";

   //Evaluate the entire command to add names to the list.
   eval $addShapesCommand;

   //Show the 9th item to force the UI to bring up a scroll bar.
   textScrollList -edit -showIndexedItem 9 blendShapeTextScrollList;

   }

}
```

The first if statement checks to ensure that there is a blendshape node actually loaded. We cannot refresh the list if nothing is loaded.

The main body of code starts by editing the text scroll list by removing all of the contents. If we did not remove them all, we may end up adding duplicates which would create a very confusing, extremely long list. To add each blendshape to the scroll list, we use the '-append' flag. Because we can never know the exact number of blendshapes in the node, we must iterate through a loop, adding blendshapes with each pass, until it no longer contains any objects. At this point, the $addShapesCommand contains a full list of all of the shapes that we need to add to the list. To actually add them, we can simply evaluate the command as done by the 'eval' MEL command.

The last line may seem confusing. The MEL text scroll list seems to have a bug whereby the scroll bar will not appear if newly added items are added after creation. To force the list to update (and thus create a scroll bar), we can tell it to show the ninth item. Because the list is only eight items long, this forces the element to create a scroll bar. Now the user can scroll through the list if it has more than eight shapes.

■ The loadDriverJoint Procedure

This procedure is called by the 'Load Driver Joint' button. It's sole purpose is to load the currently selected joint into the text field that houses the driver joint. The driver joint text field then contains the name of this joint. This allows the script to set driven keys on the joint and affect it's orientation via the joint angle slider.

```
global proc loadDriverJoint()
{
        //Get the current selection.
  string $currentSelXform[] = `ls -sl`;

  //Check to ensure that the user selected a joint.

  if (`objectType $currentSelXform[0]` != "joint")
  {

    warning "You must load a joint as a driver. ...
                          Please select a joint and try again.\n";

  } else
  {
    //Edit the text field to house the name of the driver joint.
    textField -edit -text $currentSelXform[0] driverJointTextField;
    //Provide some feedback for the user.
print ($currentSelXform[0] + " was loaded as the driver joint. \n");
  }
}
```

There really is nothing new in this procedure. This should all look quite familiar by now. The if statement checks to ensure that the first selected object is indeed of the type 'joint'. If it is a joint, the name of it is stored into the proper text field. Before exiting from the procedure, a small message is output to the feedback line to let the user know that a joint was successfully loaded. That's it.

■ The jointSliderCommand Procedure

```
global proc jointSliderCommand()
{

//Query the currently loaded joint.
string $currentLoadedJoint = `textField -q ...
                                      -text driverJointTextField`;

//Query the current value of the joint slider.
float $currentSliderValue = `floatSliderGrp -q -value jointSlider`;

//Query the current value of the joint axis radio buttons.
int $currentSelRadioButton = `radioButtonGrp -q ...
                                   -select jointAxisRadioButtons`;
//Initialize the $jointAxis string to ".r"
string $jointAxis = ".r";

//Add the proper axis name to the $jointAxis string (.rx, .ry, .rz)
      if ($currentSelRadioButton == 1)
              $jointAxis += "x";
      if ($currentSelRadioButton == 2)
              $jointAxis += "y";
      if ($currentSelRadioButton == 3)
              $jointAxis += "z";
//Check to ensure that a joint is indeed loaded.
if ($currentLoadedJoint == "")
{
  warning "You must load a joint for the slider to affect.\n";
} else
{

//Set the value of the joint's rotation to the value of the slider.
```

```
    setAttr ($currentLoadedJoint + $jointAxis) $currentSliderValue ;
  }
}
```

This procedure is called whenever the value of the joint angle slider is changed. This can happen when either the slider is dragged by the user, or the user enters a new value into the input field beside the slider. The sole purpose of this procedure is to update the currently loaded driver joint to reflect the value of the slider on the axis defined by the currently selected radio button.

Firstly, the script queries the required attributes to find the name of the current driver joint, and the current value of the joint angle slider. To find the axis to affect, we then query the radio button group to see whether X, Y or Z is selected.

The if statement at the end checks to make sure that a joint is actually loaded before using the setAttr command to set the joint's rotation value according to the queried axis and value. While dragging the slider, you may notice that your joint does not move in a perfectly fluid fashion. That is because Maya must parse and interpret this procedure several times a second to affect the joint. While not perfect, this is a nice little trick that can save a lot of setup time by preventing the user from needing to navigate through the hypergraph or viewport to find the driver joint and pose it.

■ The shapeSliderCommand Procedure

```
global proc shapeSliderCommand()
{
  //Query the currently loaded shape.
  string $currentLoadedShape = `textField -q ...
                                  -text drivenShapeTextField`;

  //Query the current value of the shape percentage slider.
  float $currentSliderValue = `floatSliderGrp -q ...
                                 -value shapePercentageSlider`;

  //Query the currently loaded blendshape node.
  string $currentBlendshapeNode = `textField -q ...
                           -text currentBlendshapeNodeTextField`;

  //Check to ensure that a shape is indeed loaded.
  if ($currentLoadedShape == "")
  {
    warning "Click on a blendshape from the list to affect it.\n";
  } else
  {
    //Set the weight of the blendshape to the value of the slider.
    setAttr ($currentBlendshapeNode + "." + $currentLoadedShape) ...
                                  ($currentSliderValue / 100) ;
  }
}
```

Like the last procedure, this one is called every time a slider is changed. This time, the slider affects a blendshape. This procedure is actually somewhat simpler though because we needn't worry about what axis is selected. A blendshape only has a weight value to worry about.

After finding the names of the currently selected blendshape node and the driven blendshape (selected from the text scroll list), we can use the setAttr command to set the blendshape's weight value. It is important to note that the value queried from the interface is a percentage (from 0 to 100). So before we applied it to a blendshape (which has a range of 0 to 1), we must divide it by 100.

■ The setShapeKey Procedure

```
global proc setShapeKey()
{

  if ((`textField -q -text drivenShapeTextField` == "") ||
     (`textField -q -text driverJointTextField` == ""))
  {
    warning "No key was set. Please load a driver joint and a ...
                              driven shape before setting a key.";
  }else
  {

    //Query the current value of the joint axis radio buttons.
     int $currentSelRadioButton = `radioButtonGrp -q ...
                                      -select jointAxisRadioButtons`;

    //Initialize the $jointAxis string to ".r"
    string $jointAxis = ".r";

  //Add the proper axis name to the $jointAxis string (.rx, .ry, .rz)
                  if ($currentSelRadioButton == 1)
                          $jointAxis += "x";
                  if ($currentSelRadioButton == 2)
                          $jointAxis += "y";
                  if ($currentSelRadioButton == 3)
                          $jointAxis += "z";

    //Build driver name string.
    string $driver = `textField -q -text driverJointTextField`;
    $driver += $jointAxis;

    //Build the driven name string.
    string $driven = `textField -q -text ...
                                      currentBlendshapeNodeTextField`;
    $driven = ($driven + "." + `textField -q ...
                                       -text drivenShapeTextField`);

    //Find the value of the driver.
    float $driverValue = `floatSliderGrp -q -value jointSlider`;

    //Find the value of the driven.
    float $drivenValue = `floatSliderGrp -q ...
                                      -value shapePercentageSlider`;
    $drivenValue = ($drivenValue / 100);

    //Set driver value.
    setAttr $driver $driverValue;
    setAttr $driven $drivenValue;
    setDrivenKeyframe -cd $driver $driven;

    print ("Key Set: (" + $driver + ", " + $driverValue + ") ...
                  and (" + $driven + ", " + $drivenValue + ") .");
  }

}
```


Once the user has adjusted the joint angle and blendshape weight, they click the 'Set Key' button to create a set driven key connection. The 'Set Key' button calls this procedure to query the current joint angle, axis and blendshape weight. With these values, the procedure can create a set driven key.

The first if statement checks to see that the user has loaded both a driver joint and a driven shape. If either of the text fields that hold this information is empty, the script returns a warning and exits.

To start with, the procedure queries the radio button group that represents the driver axis (X, Y or Z). A string variable, $jointAxis, is constructed to contain the attribute extension to specify which axis is selected as the driver (".rx", ".ry", or ".rz"). This is done with three if statements. The next few lines add the name of the joint to construct the full string (for example, jointName.rx).

To build the name of the driven shape, we must concatenate the name of the currently loaded blendshape node, with the name of the currently loaded blendshape (for example, "blendshapeNodeName" + "shapeName"). This is done by quiering the corresponding text fields and storing the result into a properly formatted string variable, $driven.

To prepare for actually setting the driven key, we must query the values of the two sliders (the blendshape and joint angle slider). These values are both stored into their own float variables, $drivenValue and $driverValue. Now we have all of the information needed to set the key.

Recall that when you set a driven key, you set the driver value first, then the driven. Then you set the key and repeat if necessary. It is no different in MEL. To do this, we use the setAttr command to set the driver and the driven attributes according to the queried results. With the attributes set, the setDrivenKeyframe command creates the actual key.

The last line is an optional piece of user feedback. It may look like a confusing mess of concatenations, but it just spits out some information about the key that was set. A typical output from this line might look like this:

Key Set: (shoulderJoint.ry, 45) and (blendShape1.shoulderFixShape, 0.55) .

That is all there is to setting driven keys with MEL.

■ The editCurve Procedure

This procedure is called when the user hits the 'Edit Curve' button. This brings up a graph editor with the currently loaded blendshape node automatically loaded. This provides a quick way for the setup artist to edit the interpolation of the set driven keys.

```
global proc editCurve()
{
        select -r `textField -q -text currentBaseMeshTextField`;
        select -addFirst `textField -q ...
                                -text currentBlendshapeNodeTextField`;
        GraphEditor;
        FrameSelected;
        print "Graph Editor Loaded.";
}
```

The default behavior of the graph editor is such that it will load whichever object was selected when it was called. So, to load the blendshape node curves into the graph, we simply select the base mesh, then the blendshape node before calling the graph editor.

To call the graph editor, we use the 'GraphEditor' MEL command. The 'FrameSelected' command will frame the graph around the first selected curve.

While we could add some error checking here, it's not really necessary. There are so many checks to ensure that the user is guided in the right direction that by the time they get to the point to edit a curve, everything should be in order and error checking at this point is extraneous.

■ The generateBlendshape Procedure

This procedure is called when the user clicks on the 'Generate Blendshape' button. The procedure requires that the user selects a sculpted mesh before clicking the button. This mesh, if created using the 'Sculpt Changes on Duplicate' button, should contain some information about the pose that it was created for. This information is stored in some custom variables (joint name and rotate channels).

The procedure starts by posing the recorded joint into the recorded pose. Then the BspiritCorrectiveShape procedure is called. This procedure actually handles the task of creating the corrected shape from the sculpted fix. After this procedure is finished, control is returned back to the the generateBlendshape procedure and we add the newly created corrected shape into the currently loaded blendshape node.

```
global proc generateBlendshape()
{
string $currentBlendshapeNode = `textField -q ...
                           -text currentBlendshapeNodeTextField`;
string $currentBaseMesh = `textField -q ...
                                -text currentBaseMeshTextField`;
string $currentSel[] = `ls -sl`;
string $currentJoint=(`getAttr($currentSel[0]+".nameOfJoint")`);

if (($currentBaseMesh == "") || ($currentBlendshapeNode == ""))
{
  warning "Please load a base mesh and a blendshape node ...
                              before generating a blendshape.\n";
}else
{

  //Prepare to call the corrective shape procedure.
  //Rotate the driver joint into the shape's pose.
  setAttr ($currentJoint + ".rx") (`getAttr ($currentSel[0] ...
                                                + ".xRotation")`);
  setAttr ($currentJoint + ".ry") (`getAttr ($currentSel[0] ...
                                                + ".yRotation")`);
  setAttr ($currentJoint + ".rz") (`getAttr ($currentSel[0] ...
                                                + ".zRotation")`);

  //Add the base mesh to the current selection.
  select -tgl $currentBaseMesh;

  //Call the procedure. Store name of new shape into string.
  string $newShape = `BSpiritCorrectiveShape`;
```

```
  $newShape = `rename $newShape ($currentSel[0] + "1")`;

  //Add the new shape to the blendshape node.
  //Using the listAttr command, we can get a list of all the shapes
  //in the node.
  string $listOfShapesInNode[] = `listAttr -k ...
                                      -m $currentBlendshapeNode`;
  int $targetIndex = 0;

  //Iterate through the list of blendshapes.
  while (!($listOfShapesInNode[$targetIndex] == ""))
    $targetIndex ++;

  //$targetIndex is now equal to the number of blendshapes.
  $targetIndex--;
  blendShape -edit -target $currentBaseMesh $targetIndex ...
                              $newShape 1   $currentBlendshapeNode;
  print ($newShape + " was added to the " ...
                   + $currentBlendshapeNode + " blendshape node.");

  if (`checkBox -q -v deleteShapeCheckBox`)
  {
    delete $newShape;
  }
 }
}
```

Up until the line that calls BspiritCorrectiveShape, this procedure does not use any new ideas. BSpiritCorrectiveShape requires that the user selects the shape and the base mesh before calling it. This procedure does that by querying the necessary text fields and using the 'select' MEL command. Notice that the line that calls the procedure captures the returned value into the $newShape variable. The BSpiritCorrectiveShape procedure returns the name of the newly created corrected shape.

The last task is to add the corrected shape into the currently loaded blendshape node. Unfortunately, editing the contents of a blendshape node is not quite as straightforward as you might expect. To do so requires that the user specify not only what shape they want to the add to what node, but also the *index* at which to add the shape. To make sense of this, think of a blendshape node as an array of shapes. To add a shape to this list, you have to tell Maya where, in the list, the shape should be added.

To find this index, I used a while loop that iterates through the blendshape node and counts each shape until the end of the list. Because the list of shapes is 0-based (like all arrays), we must subtract one from the resulting index (using the decrement notation, '--'). This leaves us with the index we can use to append the corrected shape onto the end of the list of blendshapes in the blendshape node. To actually add the blendshape, we use the 'blendshape' MEL command.

The last if statement checks to see if the user has checked the 'Delete Corrective Blendshape' check box. If they have, we delete the shape and the user never even sees the mesh before it is added to the blendshape node.

■ The BSpiritCorrectiveShape Script

OK, this is the big one. In Shape Builder, this task is called from one single procedure called BSpiritCorrectiveShape. The procedure corrects the sculpted shape by effectively subtracting displacements on the vertices that were caused by smooth skinning. To do this requires some fairly complex mathematics, the details of which are far beyond the scope of this book. While the explanation includes coverage of the mathematical principles used, the explanation of said principles is left up to the reader. If you are worried that you will not be able to follow along, I encourage the reader to take a linear algebra course and read some introductory 3d math texts. This will be adequate preparation for the following MEL lesson.

For the purposes of Shape Builder, BspiritCorrectiveShape is integrated as a single procedure. This allowed the script's functionality to be added in only a single call to a single procedure. While this single procedure might be simpler to plug into a script, it does not make for a very good example with which to teach. For this reason, I will be guiding the user through a slightly different version of the BspiritCorrectiveShape procedure. This is version 1.2 from www.b-ling.com. The advantage of using this version is that the script is clearly divided into several different procedures that are easier to understand on their own. Please realize that while the code is not exactly the same as that contained in Shape Builder, the result is the same.

Breitling's script is a fine example of how MEL can be used as a prototype for a more efficient C++ plugin. While the script is plenty fast enough for small to medium sized meshes, the algorithm is far faster when implemented through the C++ API. You can download a plugin version of the script from www.b-ling.com. With the C++ version of the corrective shape algorithm, shape correction can be computed in near realtime on meshes with over 50k vertices. This is substantially faster than the MEL version which can take a few seconds to calculate a 10k mesh. You may wonder why MEL is so inefficient. The reason has to do with the way that MEL is compiled, or rather not compiled. When a programmer writes a C++ program, the source code must be compiled (by a compiler program) to produce what is known as machine language. While this machine language usually describes an executable file, in the case of a plugin it creates a dynamically linked library (these are the .mll plugin files). These files are highly optimized and are composed of the pure binary logic that a CPU can understand without interpretation. MEL, on the other hand, is a very high level, *interpreted* language that must be parsed and interpreted at runtime (by Maya's MEL interpreter engine). This extra step is usually unnoticeable but can become quite costly when large numbers of calculations must be computed. Hence, C++ is fast and MEL is slow.

Many plugin programmers use MEL to quickly flesh out prototype algorithms. Because MEL is so close to C (syntactically), programmers find it to be an easy way to write test code. In the case of the script we are about to dissect, it is a perfect example of a deformer prototype. Learning how to write a deformer prototype can help those who are curious about venturing into the realm of C++ and plugin development. BspiritCorrectiveShape is a prime example, so let's get started!

■ Basic Program Flow

For a nice high-level view of exactly what is going on, let's talk about how this script flows. To start with, the user has two objects selected. For the duration of this lesson, these will be referred to as the base mesh (this is the bound mesh in the corrected pose) and the target mesh (this is the mesh that the user has sculpted and wants corrected). All operations involve comparing these two meshes to generate a third, corrected mesh that contains only the modeled corrections after the smooth skinning displacements are subtracted. To get from A to B, the script takes several detours in the form of calls to other procedures that complete intermediary tasks. In pseudo-code the whole process looks like this:

```
BSpiritCorrectiveShape Procedure ()
{

  get name of $base and $target meshes;
  get the vertex count for both meshes;

  for (every vertex)
  {

    call BSpiritCorrectiveShapePosition ()
    {

      get the world position for base and target;
      calculate the distance between base and target;
      return distance;

    }
    if distance > 0 //This means the vertex was sculpted.
    {

      call BspiritCorrectiveShapeVectorMove()
      {

        //Find basis vectors for vertex space
        move base vertex +1 in local X;
        record base vertex new world position;
        move base vertex back to original position;
        move base vertex +1 in local Y;
        record base vertex new world position;
        move base vertex back to original position;
        move base vertex +1 in local Z;
        record base vertex new world position;
        move base vertex back to original position;

        //Do calculations
        calculate new base vertex displacement vector;
        move base vertex by calculated vector;

      }

    disable all deformers on the base mesh;
    duplicate the base mesh to create the corrected shape;
    rename the duplicate;
    reset original base vertex positions;
    re-enable all deformers on the base mesh;

    }

  }

}
```

The script finds the number of vertices in the base mesh and creates a for loop to iterate through every vertex. Remember that each vertex in a mesh is numbered, like indices in an array. This is the basic structure for every deformer.

For every vertex, the program checks to see if the modeler has sculpted a change on it. It does this by comparing each vertex on the base mesh to it's corresponding vertex on the target. By taking into consideration any transformations, the script is able to effectively superimpose the target mesh overtop of the base mesh. The distance is calculated between each corresponding vertex. If this distance is greater than zero, then the vertex was moved and it's displacement vector must be calculated. If the distance is zero, the loop skips the current vertex and moves on. This prevents the program from doing unnecessary calculations on vertices that do not need it. In fact, without this check, the program would be quite inefficient.

In the case that the vertex has been moved, the program calls the BspiritCorrectiveShapeVectorMove procedure. This procedure is where the magic happens. To understand why this procedure does what it does, we need to first step back and study the problem on a more abstract level.

When you bind a mesh to a skeleton, Maya creates a skinCluster node. This node is actually just a special deformer. The skinCluster deformer calculates how much to move a vertex based on the sum of it's weighted influence transformations. What is more important is that it changes the way the vertices behave when you try to move them. You can witness this yourself with a very simple example. Try smooth binding a primitive object to a simple joint chain. After posing the skeleton, it becomes impossible to sculpt the mesh in any normal way. For example, translating a vertex in positive X (using the regular translate manipulator), can cause the vertex to move off in a seemingly arbitrary direction! It's like Maya is acting drunk. What seems like a simple single-axis translation can result in the vertex sliding off in a completely different direction. This behavior is the result of the skinCluster putting each vertex into a different coordinate space.

So, the BspiritCorrectiveShapeVectorMove procedure must find some way of being able to move the bound vertex to match the corresponding vertex on the target mesh. Because the vertices are in different coordinate spaces (and usually not even close to world space), the displacement vector must be calculated to move vertex A to point B in world space using the local coordinate space. This is done using a coordinate space transformation. Linear algebra provides several different methods of calculating a displacement vector for an arbitrary coordinate space.

■ The Math Principles

If you take any number, it can be multiplied by 1 to arrive at the same number (5 x 1 = 5, -4596.23432 x 1 = -4596.23432). Vectors have a similar principle. For every vector (a), there exits a square matrix (M), such that when M transforms a, the resulting vector (b) is equal to a. More formally:

$$aM=b$$
$$a=b$$

For a vector in world space, this matrix looks like this:

```
| 1 0 0 |
| 0 1 0 |
| 0 0 1 |
```

If the vector's coordinate space is different than the world space (as is the case with vertices bound to a skeleton), the square matrix could look quite different. It might look something like this:

```
| -4 2 1 |
| 1 3 -2 |
| 7 7 -10|
```

Each row of the matrix above can be interpreted as the basis vectors of a coordinate space. So the basis vectors for this space are [-4 2 1], [1 3 -2], and [7 7 -10]. This means that if you multiply a vector in this space by the matrix above, you will be left with the same vector.

To find the basis vectors of the current vertice's coordinate space, the script translates each vertex one unit in each direction (x, y, and z) and records the new world space position of the vertex after each change in position. From this information, we can derive a matrix that can transform the vertex to any world space vector. In this case, we want to translate each base mesh vertex to the position of the corresponding vertex on the target mesh. The problem then is to determine the *local* displacement vector that will translate the vertex into the proper *world* space position. To see how this is done, let's consider an example:

We have vertex P that needs to be translated to the world space position [3,9,-14]. To find the local space vector that will translate this vertex, we need to find the basis vectors for this local coordinate space. We start by moving the vertex by one unit in each local direction and recording the world space position after each displacement:

1. Move P by 1 unit in local X (1,0,0): This positions the vertex in world space (-4,2,1).
2. Move P by 1 unit in local Y (0,1,0): This positions the vertex in world space (1,3,-2).
3. Move P by 1 unit in local Z (0,0,1): This positions the vertex in world space (7,7, -10).

Now we have enough information, we could move the vertex to a world space position like [-8,2,2] quite easily ([-8,2,2,] is [-4,2,1] x 2). To move the vertex to this position, we would displace it by 2 units in X (2,0,0). We can calculate the displacement vector for any multiple of the basis vectors using this technique.

That's all fine and dandy, but we need to move our vertex to a very specific position (the target vertex world space position). Let's say P's target vertex was located at the world space position [3, 9, -14]. How do we move P to this world space position? This requires that we solve a system of equations. For this case, the linear system of equations looks like this:

```
-4x + y + 7z = 3
2x + 3y + 7z = 9
1x + -2Y -10z = -14
```

Solving for x, y and z will give us the local displacement vector that will translate P into the world position [3, 9 ,-14]. Solving systems of equations like this can be done in many different ways. The most popular way to solve a system of equations like this is using a technique known as Gaussian elimination. While Gaussian elimination works fine, it is computationally inefficient. This script uses a method known as Cramer's rule and is quite popular for solving these kinds of problems. Solving this system results in the vector [-3, 2, 2]. By moving P -3 units in X, 2 in Y and 2 in Z (in local space), it will be in the world space position [3,9,-14].

This calculation is done for each vertex in the BSpiritCorrectiveShape-VectorMove procedure.

■ The Code

Now that we have an understanding of exactly what the script does, let's take a look at how it works starting with the declaration of the main procedure.

```
global proc BSpiritCorrectiveShape() {

// Declaring main variables & Query names and selections

  string $SelectedVertexArray[] = `filterExpand -sm 31`; // Vertex
  // All - with hierarchy
  string $SelectedItemArray[] = `ls -selection -long`;
  string $SelectedMeshArray[] = `filterExpand -sm 12`; // Polygon
  string $BaseModel;
  string $BlendShape;

  int $SelectedVertexNumberArray[];
  int $SelectionTrue = 0;
  int $BREAK = 0;
  int $FailureList = 0; // [0: Everything OK]
                        // [1: Selection is false]
                        //[2: Skin is missing]
                        //[3: User abort]
                        //[4: PolyCount not identical]
                        //[5: Nothing to Move]

int $VertexCount = `size $SelectedVertexArray`;
int $ItemCount = `size $SelectedItemArray`;
int $MeshCount = `size $SelectedMeshArray`;

if (($VertexCount > 0) && ($MeshCount == 1)) {

  $SelectionTrue = 1;
  string $Temp[];
  $BaseModel = $SelectedItemArray[($ItemCount - 1)];
  tokenize $SelectedItemArray[0] "." $Temp;
  $BlendShape = $Temp[0];
  tokenize $SelectedVertexArray[0] "[" $Temp;
  int $TempPrefixSize = `size $Temp[0]` + 2;

  for ($i = 0; $i < ($VertexCount); $i++) {

    string $TempString = $SelectedVertexArray[$i];
    int $TempSuffixSize = `size $TempString` - 1;
    $SelectedVertexNumberArray[$i] = `substring ...
                    $TempString $TempPrefixSize $TempSuffixSize`;

  }
}

else if (($MeshCount == 2) && ($ItemCount == 2)) {

  $BlendShape = $SelectedItemArray[0];
  $BaseModel = $SelectedItemArray[1];
```



```
}

else {

  $BREAK = 1;
  $FailureList = 1;

}
```

The script starts out by storing the initial selections of vertices and meshes. This version of the script allows the user to specify a selection of vertices to run the script on. If the user has some vertices selected, the script will only act on these vertices and ignore any unselected points. This can provide some better performance for meshes with a very large number of vertices.

Notice the use of the 'filterExpand' flag. This provides a way for the program to filter out different types of selections. In this case, we have the '-sm' (-selectionMask) flag followed by the number 31. The number 31 corresponds to polygonal vertices. So, '-sm 31' tells Maya to mask the selection to contain only polygonal vertices. The returned value is then stored into $SelectedVertexArray. There is a full list of 73 different ways to filter or expand your current selection (found in the MEL command reference). These filters can be very useful.

The rest of the variables in the initial declaration are self explanatory. You may be wondering about the $FailureList int value. This script uses a nifty trick to track the different kinds of errors that may be encountered. By setting this value to 0, the script is in an 'error-free' state. As the script checks for different errors, this variable could be assigned to reflect what went wrong. The script includes comments that explain what the different error codes mean.

The $VertexCount variable holds an integer value that represents the number of vertices that the user selected before executing the script. Likewise, the $ItemCount and $MeshCount variables house the number of items and meshes in the initial selection. Notice the use of the 'size' function. This is a great MEL function for finding the number of items in an array or the number of characters in a string.

The first if statement is used to determine if the user has selected any vertices to specify a calculation mask. If the $vertexCount variable is not equal to 0, it means that the user has selected some points on their mesh. In the case that this if statement evaluates to true, a flag ($SelectionTrue) is set to 1. The $BaseModel variable houses the name of the base mesh which is currently stored in the last index of the $SelectedItemArray.

The next line uses the tokenize MEL command. The command is used to split up the first string wherever there is an instance of the specified character. In this case, the split character is a '.'. If no split character is specified, the tokenize command will split up the string wherever it encounters a whitespace. The last argument is a string array that houses the results of the splitting.

The for loop is used to load-up the $SelectedVertexNumberArray until it holds the number of every vertex that the user selected. It has to extract these numbers from the full list of vertices stored in $SelectedVertexArray. These selected vertices are stored in the form 'nameOfMesh.vtx[23]' where 23 could be any number representing an individual vertex on the mesh. To extract the number from between the square brackets, the for loop uses the substring and size functions to manipulate the array's contents and store the needed vertex numbers.

In the case where the if statement that checks for selected vertices is skipped (in the case that the user decided not to specify certain vertices for calculation), the script checks to ensure that the user has selected only the base and target meshes. If the selection is correct, the name of the base and target meshes are stored into their corresponding string variables. In the case that the selection is incorrect, the script sets the $BREAK variable to 1 (which causes the main loop to exit) and sets the error code variable, $FailureList, to 1. The error code 1 will return a warning later in the script to inform the user that the selection was incorrect.

After ensuring that the user has the correct selection number and type, we can get to work on figuring out what skinCluster and tweak node we will be working with.

```
string $TweakNode;
string $Skin[];

  float $Progress = 0.0;

  if (! $BREAK) {

    string $BaseHistory[] = `listHistory`;

    int $HistorySize = `size $BaseHistory`;

    for ($i = 0; $i < $HistorySize; $i++) {

      string $TempHistory = `nodeType $BaseHistory[$i]`;

      if ($TempHistory == "tweak")
        $TweakNode = $BaseHistory[$i];
      else if ($TempHistory == "skinCluster")
        $Skin[`size $Skin`] = $BaseHistory[$i];

    }

    if ($TweakNode == "" || $Skin[0] == "") {

      $BREAK = 1;
      $FailureList = 2;

    }
  }
```

This entire code block is skipped if $BREAK is equal to 1 (this indicates that the user has an erroneous selection). The $BREAK variable is an excellent example of how to track errors across a series of mini-tasks and different procedures.

This next little loop is actually quite interesting. This is a fine example of how to find related dependency graph nodes. The $BaseHistory array is initialized to contain the name of every dependency node connected to the base mesh. Recall that skinClusters and tweak nodes reside in the dependency graph, just like everything else. The 'listHistory' function returns a full list of related nodes.

Now we enter into the for loop to iterate through each related node

and determine whether or not it is of type skincluster or tweak (it may be something else but we are only concerned with finding these two nodes).

The last code block (outside of the loop) checks to make sure that a skinCluster and a tweak node were both found. If either of said nodes are missing, the $FailureList variable is set to error code 2 and the $BREAK variable is set to 1. Error code 2 will return an error (at the end of the script) that warns the user about a missing skin.

With the related nodes found, we are ready to initialize the progress window and gather the last bits of information needed for the main loop.

```
float $Offset[3];

int $BaseVertexCount[];

string $vertexNameArray[];
string $TweakVertexArray[];
string $BaseModellVertexNameArray[];

if (! $BREAK) {

  progressWindow

    -title "BSpirit Corrective Shape"
    -progress $Progress
    -status "Calculating VertexMovements"
    -isInterruptable true
    -min 0
    -max 100;

  float $BaseTempOffset[3];
  float $BlendTempOffset[3];

  $BaseVertexCount = `polyEvaluate -v $BaseModel`;
  $BaseTempOffset = `getAttr ($BaseModel+".translate")`;
  $BlendTempOffset = `getAttr ($BlendShape+".translate")`;

  $Offset[0] = $BlendTempOffset[0] - $BaseTempOffset[0];
  $Offset[1] = $BlendTempOffset[1] - $BaseTempOffset[1];
  $Offset[2] = $BlendTempOffset[2] - $BaseTempOffset[2];


  int $BlendVertexCount[] = `polyEvaluate -v $BlendShape`;

  string $ShapeNode[] = `pickWalk -d down $BaseModel`;

  if (! `connectionInfo ...
              -isDestination ($ShapeNode[0] + ".tweakLocation")`)
    connectAttr ($TweakNode + ".vlist[0].vertex[0]") ...
                              ($ShapeNode[0] + ".tweakLocation");

   if ($BaseVertexCount[0] != $BlendVertexCount[0]) {

    $BREAK = 1;
    $FailureList = 4;

  }
}
```

To start with, we declare the $Offset array to house the world space offset (if the target mesh is not coincident with the base mesh). The rest of the variables are self explanatory and will soon house important information for the rest of the script.

Before we bring up the progress window, we check to see that the $BREAK variable is not equal to 1. Recall that this would be indicative of having encountered an error already. With no errors, we can create a progress window. Progress windows are a great way to show the user how long the script is going to take to complete it's task. Personally, I hate waiting for Maya to complete a task with no visual representation of exactly how long it is going to take. If the user waits 5 seconds before seeing the progress of an executed command, they immediately begin to assume it has gone awry. A progress window is soothing relief that reassures the user that your program hasn't completely hijacked their Maya session.

A progress window works by first being created. To show the progress (in the form of blue bars), you must edit the window's '-progress' attribute. Calculating this progress is left up to the programmer, but it usually involves factoring-in the state of a loop and editing the progress at each iteration of the loop. This way, the progress bar reflects the state of the main loop and the overall time it will take to complete the task.

After initializing the progress window, we calculate the world space offset between the base mesh and the target mesh. This is done by querying the translate attributes and storing these into the $BaseTempOffset and $BlendTempOffset variables. The difference between the corresponding x, y and z components are stored into the $Offset array. It is interesting to note that this method of calculating an offset is dependant on the user having not frozen the transformations on the blendshape. If they did, the translate values will read '0' and no offset will be recorded. Setup artists should be accustomed to not freezing transforms on their blendshapes, so this is not really an issue.

After calculating the offset, the script goes on to find the number of vertices (used for the loop) by calling the 'polyEvaluate' MEL command. The 'polyEvaluate' command is essential for finding information about a polygonal mesh. The $ShapeNode array is initialized to the name of the shape node attached to the base mesh (this is found with pick walking).

The first if statement checks to see if the tweak node is connected to the base mesh. There is a Maya bug that may cause a tweak node to become separated (this can happen if a smooth filter is applied to the mesh). Regardless, vertex movements will not function correctly if the connection is severed.

Lastly, the script checks to make sure that the number of vertices in the base mesh match the number of vertices in the target mesh. If these meshes have a different topology, the script sets $BREAK to 1 and $FailureList to 4. The error code '4' returns an error at the end of the script that warns the user about a difference between the base and target meshes.

The next part of the script is responsible for finding three important pieces of information stored into three different float arrays. The $targetPosArray stores the local position of each vertex on the target mesh. The $absPosArray stores the world position and the $relPosArray stores the world displacement vector for the base to reach the target position.

```
float $targetPosArray[];
float $absPosArray[];
float $relPosArray[];

if (! $BREAK) {

  if (! $SelectionTrue) {

    $VertexCount = $BaseVertexCount[0];

  }

  for ($i = 0; $i < $VertexCount; $i++) {

    if ( `progressWindow -query -isCancelled` ) {

      $BREAK = 1;
      $FailureList = 3;
      break;

    }

    int $LastProgress = $Progress;
    $Progress = `trunc (20.0 * $i / $VertexCount + 0.5)`;

    if ($LastProgress != $Progress)
      progressWindow -e -progress $Progress;

    string $VertexAppendix;

    if (! $SelectionTrue)
      $SelectedVertexNumberArray[$i] = $i;

    $VertexAppendix = ".vtx[" + $SelectedVertexNumberArray[$i]...
                                                    + "]";

    string $BaseModelVertexName = $BaseModel + $VertexAppendix;
    string $BlendShapeVertexName = $BlendShape + $VertexAppendix;

    float $PositionArray[10];

    $PositionArray = `BSpiritCorrectiveShapePosition ...
                  $BlendShapeVertexName $BaseModelVertexName ...
                                $Offset[0] $Offset[1] $Offset[2]`;

    if ($PositionArray[0] == 1) {

      int $vertexArraySize = `size $vertexNameArray` * 3;

      $vertexNameArray[$vertexArraySize / 3] = ...
                                              $BaseModelVertexName;
      $TweakVertexArray[$vertexArraySize / 3] = ...
$TweakNode + ".vlist[0].vertex[" + $SelectedVertexNumberArray[$i];

      $absPosArray[$vertexArraySize] = $PositionArray[1];
      $absPosArray[$vertexArraySize + 1] = $PositionArray[2];
      $absPosArray[$vertexArraySize + 2] = $PositionArray[3];

      $targetPosArray[$vertexArraySize] = $PositionArray[4];
      $targetPosArray[$vertexArraySize + 1] = $PositionArray[5];
      $targetPosArray[$vertexArraySize + 2] = $PositionArray[6];

      $relPosArray[$vertexArraySize] = $PositionArray[7];
      $relPosArray[$vertexArraySize + 1] = $PositionArray[8];
      $relPosArray[$vertexArraySize + 2] = $PositionArray[9];
    }
  }
}
```

If the $BREAK variable is equal to '0', the script starts by determining whether or not the $SelectionTrue variable is true or false. If true (equal to 1), the vertex count is from the user selection, otherwise we iterate through every vertex in the mesh. The for loop here iterates through every vertex. It starts off by calculating the current progress and editing the progress window to reflect this. Notice that the script does a quick check to make sure that progress has been made since the last iteration. This prevents the progress window from updating unnecessarily, thus slowing down the whole process.

After constructing the name of the base and target vertex (of the form meshName.vtx[$i]), the script sends this information off to the BspiritCorrectiveShapePosition procedure. This procedure does the necessary queries and returns a value of 1 if the base and target vertices are not coincidental (in the same position). It also returns the relative and world positions as well as the world space displacement vector to move the base vertex to the target position.

The return values are stored into the three major arrays ($absPositionArray, $targetPositionArray, and $relPositionArray. We now have enough information to begin computing the local displacement vector that will move our base mesh vertex into the target position.

Before we do coordinate space transformations, we can check to see exactly how many of the vertices were sculpted out of position. The next block of code checks to see if more than half of the meshes vertices were moved. If this is the case, we warn the user about high vertex movements (because this number of moved vertices would be very unlikely).

```
int $vertexNameArraySize = `size $vertexNameArray`;
int $PercentMoved;

if ($vertexNameArraySize > 0)
  $PercentMoved = `trunc (0.5 + ...
                    $vertexNameArraySize / $VertexCount * 100)`;

else {

  $BREAK = 1;
  $FailureList = 5;

}


if ($PercentMoved > 50) {

  if (`confirmDialog

      -title "Moving Vertex"
      -message ("About " + $PercentMoved ...
                          + "% vertices will be moved. Proceed?")
```

```
                    -button "Proceed" -button "Abort"
                    -defaultButton "Proceed" -cancelButton "Abort"
                    -dismissString "Abort"`== "Abort")
        {
      $BREAK = 1;
      $FailureList = 3;

    }

}
```

This code block starts by checking to ensure that there are indeed some vertices that need to be moved. If there are no moved vertices, the script sets the $BREAK variable and records an error code of 5 into the $FailureList. Error code 5 returns a warning that there are no vertices to move.

If the target mesh has over half of it's vertices moved, the user is confronted with a confirm dialog box that asks if this the script should continue. A high number of moved vertices could be indicative of a user error and may take a very long time to calculate. The dialog box is a final check to make sure that the user is sure about their target mesh. If the user cancels at this point, the $BREAK variable is set and an error code of 3 is recorded into the $FailureList variable. Error code 3 will return a warning that the user has aborted the script process.

The next section of code does the actual moving of the vertices on the base mesh to match to the target mesh.

```
if (! $BREAK) {

    progressWindow -e
       -status ("Moving " + $vertexNameArraySize + " vertices");


for ($i = 0; $i < $vertexNameArraySize; $i++) {

  if ( `progressWindow -query -isCancelled` ) {

    $BREAK = 1;
    $FailureList = 3;
    break;

  }


  int $LastProgress = $Progress;
  $Progress = `trunc (20.0 + 60.0 * $i / ...
                                $vertexNameArraySize + 0.5)`;

  if ($LastProgress != $Progress)
    progressWindow -e -progress $Progress;

  int $ArrayPos = $i * 3;

  BSpiritCorrectiveShapeVectorMove
            $TweakVertexArray[$ArrayPos / 3]
            $vertexNameArray[$ArrayPos / 3]
          $absPosArray[$ArrayPos] $absPosArray[$ArrayPos + 1]
       $absPosArray[$ArrayPos + 2] $targetPosArray[$ArrayPos]
$targetPosArray[$ArrayPos + 1]  $targetPosArray[$ArrayPos + 2]
   $relPosArray[$ArrayPos]
$relPosArray[$ArrayPos + 1]     $relPosArray[$ArrayPos + 2];

    }
  }
```

We start out by checking the status of the $BREAK variable. If everything is in order, the script begins by updating the progress window to display a different status. The status of a progress window is the little message that is printed above the progress bar to tell the user what part of the process the script is currently working on.

We then enter into a loop that iterates through every vertex that we have determined to be in need of being moved. For each vertex, the script calculates and updates the progress window to reflect the current value of $Progress. The $Progresss variable is computed as the truncated product of the current index divided by the number of total iterations. This value is edited to fall into a range that represents the amount of the total progress that this procedure takes up.

Finally, the BspiritCorrectiveShapeVectorMove procedure is called to calculate and move the each vertex into it's proper position. The list of arguments reflects the information needed during these calculations. This information has been collected of the course of main procedure and includes the pertinent position values for the current vertex. If the script were to quit at this point, the base mesh would look exactly like the target mesh. All the vertices are no lined up, the only thing left to do, is subtract the smooth skinning displacements from the base mesh. This is as simple as disabling the deformers on the base mesh (most importantly the skinCluster deformer).

```
string $AllInputs[] = `listHistory $BaseModel`;
string $MutedTypes[] = {"skinCluster", "blendShape"};


 if (! $BREAK) {

   for ($i = 0; $i < `size $AllInputs`; $i++){
     for ($z = 0; $z < `size $MutedTypes`; $z++){

     if (`nodeType $AllInputs[$i]` ==  $MutedTypes[$z])
       setAttr ($AllInputs[$i] + ".nodeState") 1;

     }
   }

  string $BlendDuplicate[] = `duplicate -rc -rr $BaseModel`;
  string $UnlockAttributes[] = ...
         {".tx",".ty",".tz",".rx",".ry",".rz",".sx",".sy",".sz"};

   for ($i = 0; $i < 9; $i++) {

     setAttr -lock 0 ($BlendDuplicate[0] + $UnlockAttributes[$i]);

   }

   select -r $BlendDuplicate[0];

 }
```

To start with, we find all of the inputs on the mesh. This is done with the 'listHistory' function. Then we make a list of deformers that we want to disable. This version of the script disables both skinCluster and blendshape deformers. This allows you to sculpt corrective shapes on top of other corrective shapes!

To disable the inputs, we iterate through each input (as gathered in the $AllInputs array), and if the input is of type 'skinCluster' or 'blendShape', we set it to 'Has No Effect'. This is done by setting it's '.nodeState' attribute to 1. This effectively stops the input from affect the mesh. In this case, it subtracts the skeleton-based deformation leaving only the sculpted changes.

The base mesh now looks like a corrected shape. To make the corrected shape, we duplicate the base mesh. The script then unlocks all of the transforms (recall that a smooth bound mesh has it's transforms locked). And to finish it off, we select the newly created, corrected shape for the user.

While the script has finished what it set out to do (create a corrected shape), we aren't quite finished yet. At this point, the base mesh has had it's skinCluster and blendShape nodes set to 'Has No Effect'. We need to reset the base mesh and remove the displacements on all of the vertices.

```
progressWindow -e -st "Reseting BaseMesh";

  for ($i = 0; $i < $vertexNameArraySize; $i++) {

    int $LastProgress = $Progress;
    $Progress = `trunc (80.0 + 20.0 ...
                                  * $i / $vertexNameArraySize + 0.5)`;
    if ($LastProgress != $Progress)
      progressWindow -e -progress $Progress;

    int $ArrayPos = $i * 3;

    setAttr ($TweakVertexArray[$ArrayPos / 3] ...
                              + "].xVertex") $relPosArray[$ArrayPos];
    setAttr ($TweakVertexArray[$ArrayPos / 3] ...
                            + "].yVertex") $relPosArray[$ArrayPos + 1];
    setAttr ($TweakVertexArray[$ArrayPos / 3] ...
                           + "].zVertex") $relPosArray[$ArrayPos + 2];

  }


  for ($i = 0; $i < `size $AllInputs`; $i++){
    for ($z = 0; $z < `size $MutedTypes`; $z++){

    if (`nodeType $AllInputs[$i]` ==  $MutedTypes[$z])
      setAttr ($AllInputs[$i] + ".nodeState") 0;

    }
  }

  progressWindow -endProgress;
```

To start off with, this block of code sets the progress window to reflect the current task. It then continues on with setting the base mesh vertices back to their original positions. Finally, the nested for loops set the skinCluster and blendShape input's .nodeState attributes to 'o'. This enables the muted deformers.

Finally, we close the progress window using the -endProgress flag. At this point, the script has completed it's primary objective by creating the corrected shape and setting the base mesh back to it's original position. To finish things off, we provide some feedback to the user. This final block of code determines which, if any, errors have been returned and pops up a dialog box to warn the user about the specific error encountered.

```
if ($FailureList != 0) {

  string $Feedback;
  switch ($FailureList) {

    case 1:
      $Feedback = "Wrong number of objects selected!";
      break;
    case 2:
      $Feedback = "Second mesh has no skinCluster!";
      break;
    case 3:
      $Feedback = "Procedure aborted by user!";
      break;
    case 4:
      $Feedback = "Not identical amount of vertices!";
      break;
    case 5:
      $Feedback = "Nothing to move!";
      break;
    }

    confirmDialog   -title "Failure - Procedure aborted"
                    -message $Feedback
                    -button "OK";

  }
}
```

That is the completed BSpiritCorrectiveShape procedure. Next, we will take a look at the two other procedures that are called from the main procedure. Let's start with the procedure that is called to query the positions of the vertices.

■ BSpiritCorrectiveShapePosition

This procedure is called to return the local and world position of the current base vertex. It also returns a world space displacement vector to reach the target position as well as a distance indicator. This distance indicator is equal to o if the base and target vertex are in the same position and the vertex is skipped in subsequent calculations. If the target vertex was moved, it returns 1 and must be corrected later in the script.

```
proc float[] BSpiritCorrectiveShapePosition( string
$BlendShapeVertexName, string $BaseModelVertexName, float $OffsetX,
float $OffsetY, float $OffsetZ)

{
```

```
  float $targetPos[3];
  float $Pos[3];
  float $relVertexPos[3];

  $targetPos = `pointPosition -w $BlendShapeVertexName`;
  $Pos = `pointPosition -w $BaseModelVertexName`;
  $relVertexPos = `getAttr $BaseModelVertexName`;

  $targetPos[0] -= ($Pos[0]+$OffsetX);
  $targetPos[1] -= ($Pos[1]+$OffsetY);
  $targetPos[2] -= ($Pos[2]+$OffsetZ);

  if (($targetPos[0]>0.001) || ($targetPos[0]<-0.001) ||
      ($targetPos[1]>0.001) || ($targetPos[1]<-0.001) ||
      ($targetPos[2]>0.001) || ($targetPos[2]<-0.001))

    return { 1.0,
             $Pos[0],          $Pos[1],          $Pos[2],
             $targetPos[0],    $targetPos[1],    $targetPos[2],
             $relVertexPos[0], $relVertexPos[1], $relVertexPos[2] };

  else

    return { -1.0 };

}
```

■ BSpiritCorrectiveShapeVectorMove

This procedure is called to calculate the local displacement vector that will move the base vertex into the target vertex position. It does this using a method of solving a series of equations known as Cramer's Rule. Notice that while matrix math is being computed, the actual matrix data type is not used. Many programmers find it easier to use simple float arrays as opposed to the matrix data type. The reason for this being that the matrix data type does not support an sort of operator overloading. All calculations must be computed explicitly. The only advantage they have, if any, is that of convenient organization of the matrix data. Regardless of what you use, the calculations must be done manually.

```
proc BSpiritCorrectiveShapeVectorMove( string $TweakVertexName,
string $BaseModelVertexName,float $WorldPosX,  float $WorldPosY,  float
$WorldPosZ,float $targetPosX, float $targetPosY, float $targetPosZ,float
$relPosX, float $relPosY, float $relPosZ )

{
  float $tempPos[3];
  float $Matrix[12];

  setAttr ($TweakVertexName + "].xVertex") ($relPosX + 1);
  $tempPos = `pointPosition -w $BaseModelVertexName`;
  $Matrix[0]  = $tempPos[0]-$WorldPosX;
  $Matrix[4]  = $tempPos[1]-$WorldPosY;
  $Matrix[8]  = $tempPos[2]-$WorldPosZ;
  $Matrix[3]  = $targetPosX;

  setAttr ($TweakVertexName + "].xVertex") ($relPosX);
  setAttr ($TweakVertexName + "].yVertex") ($relPosY + 1);
```

```
$tempPos = `pointPosition -w $BaseModelVertexName`;

$Matrix[1]  = $tempPos[0]-$WorldPosX;
$Matrix[5]  = $tempPos[1]-$WorldPosY;
$Matrix[9]  = $tempPos[2]-$WorldPosZ;
$Matrix[7]  = $targetPosY;


setAttr ($TweakVertexName + "].yVertex") ($relPosY);
setAttr ($TweakVertexName + "].zVertex") ($relPosZ + 1);

$tempPos = `pointPosition -w $BaseModelVertexName`;

$Matrix[2]  = $tempPos[0]-$WorldPosX;
$Matrix[6]  = $tempPos[1]-$WorldPosY;
$Matrix[10] = $tempPos[2]-$WorldPosZ;
$Matrix[11] = $targetPosZ;

float $return[3];

float $Denominator =

($Matrix[0]*(($Matrix[5]*$Matrix[10])-($Matrix[6]*$Matrix[9]))) -
($Matrix[1]*(($Matrix[4]*$Matrix[10])-($Matrix[6]*$Matrix[8]))) +
($Matrix[2]*(($Matrix[4]*$Matrix[9] )-($Matrix[5]*$Matrix[8])));

if ($Denominator != 0) {

  $return[0] =


(
($Matrix[3]*(($Matrix[5]*$Matrix[10])-($Matrix[6]*$Matrix[9]  ))) -
($Matrix[1]*(($Matrix[7]*$Matrix[10])-($Matrix[6]*$Matrix[11] ))) +
($Matrix[2]*(($Matrix[7]*$Matrix[9])-($Matrix[5]*$Matrix[11] )))
 ) / $Denominator;



  $return[1] =

 (
 ($Matrix[0]*(($Matrix[7]*$Matrix[10])-($Matrix[6]*$Matrix[11]))) -
 ($Matrix[3]*(($Matrix[4]*$Matrix[10])-($Matrix[6]*$Matrix[8]))) +
 ($Matrix[2]*(($Matrix[4]*$Matrix[11])-($Matrix[7]*$Matrix[8])))
 ) / $Denominator;



  $return[2] =

 (
 ($Matrix[0]*(($Matrix[5]*$Matrix[11])-($Matrix[7]*$Matrix[9]))) -
 ($Matrix[1]*(($Matrix[4]*$Matrix[11])-($Matrix[7]*$Matrix[8]))) +
 ($Matrix[3]*(($Matrix[4]*$Matrix[9])-($Matrix[5]*$Matrix[8])))
 ) / $Denominator;
```

```
    setAttr ($TweakVertexName + "].xVertex") ...
                                            ($relPosX + $return[0]);
    setAttr ($TweakVertexName + "].yVertex") ...
                                            ($relPosY + $return[1]);
    setAttr ($TweakVertexName + "].zVertex") ...
                                            ($relPosX + $return[2]);

  }
}
```

Notice that after computing the local x, y and z displacements, the script uses the setAttr command to move each vertex. This is the basic functionality of all deformers. Calculate how you want to move a vertex, then move it. This is repeated for every vertex on the mesh (or in this case the vertices specified by the user).

Final Thoughts:

As you can see, Shape Builder is definitely the most complicated script we've encountered thus far. The modular approach allowed us to plug in different functionality to build up a hierarchy of control flow. This is a prime example of just how far you can take MEL. Even if you didn't fully understand all of the math or computations, I hope that you stuck through it and took away everything that you could. Shape Builder provides some great lessons about error checking, structured programming, and control flow.

Remember, reading someone else's code is, by far, the most difficult task in programming. Believe it or not, writing complicated scripts is far easier than trying to read and understand one. The reason for this is simple. When you write a script, you can add exceptions and particular cases to account for errors *as they arise*. While a certain amount of foresight is undoubtedly necessary, trying to determine every single procedure and conditional that will be necessary is an exercise in futility. Programming is half art, half science. A finished program is not the result of strict adherence to a formula, but rather an evolved system of logic that flows from the trial and error of development.

That is not to discourage you from reading other's code. Indeed, this can be the best way to learn the specifics of how to handle certain situations. Trying to understand other people's code can lead you to revelations about the way you handle certain programming problems yourself. What I'm trying to say is, study other's code, but don't get discouraged if you aren't able to read it like a book. This stuff is hard!

INDEX

Absolute Name, 30
Action Unit, 97
Architecture, 4
Argument, 25,29
arrays, 164
ASCII, 24
Auto-Forearm Rig, 21
Automatic Type Declaration, 29
Auto-rigger, 28
back quotes, 150
back ticks, 150
Blendshape, 89, 116, 117
Boolean, 33
break, 170
button, 152
catch (scripting), 206
Channels, 11
Cleaning a Mesh, 95
Cleaning, 11
Collision, 80
Colors (Control Objects), 11
Commands (Buttons), 152
Commands, 25
comments, 147, 148
Concatenate, 29, 170, 171
Conditional, see if statements
Connecting Control Objects, 10
continue, 170
strings, (building), 170-171
coordinate space transformation, 225, 226
corrective shapes, 227
Cramer's Rule, 244
Cycle, 5
DAG, 4
Dampening, 59
Data Types, 26
Deformation Order, 42-44, 121
deformations, 212
Dependancy Graph, 4
DG, 4
displacements (vector), 117
Distance tool, 38, 95
dot product, 224, 225
Drag, 59
else, 168
End Effector, 23
end terminator, 149
Error Checking, 28, 80, 237
escape character, 149
Expressions, 31, (scripting) 128 - 129
feof (end of file), 184
file browse dialog, 201
file formats, 130
files, 130-132, 137, 168
filter selection, 245
Fingers Joints, 8
Flags, 25
Flipping, 31
Float, 26, 149
for loop, 173
for-in loop, 165
Forward Kinematics, 9
Freeze Transformations, 95
Friction, 59
Gaussian Elimination, 244
Global Procedures, 28
Goal weights, 65
Gravity, 59
Ground Plane, 95
Grouping, 19, 33
Hair, 58
Hierarchy of Control, 19
Hypergraph, 4, 12
Iconic Representation, 9-11
If statements, 29, 150
IK/FK Switch, 17
influence objects, 213
Integer, 26, 149
Inverse Kinematics, 9
Joints, 5
keyframe, 175
Knee
Layers, 12-13
Local Rotation Axis (Joints), 6, 8, 46
log files (scripting), 205
loops, 165
Macro, 25, 27
Matrix, (math) 226
MEL, What is It? 24
Merge Vertices, 95
Mirroing, Blendshapes, 116
Mirroring Joints, 8-9
multi-axis driver, 221
multiplyDivide, 41,42
Naming Joints, 9
nested loops, 174
No-Flip Knee, 13
Normalization, 120
Normals (Smoothing), 95
Obj File Format, 95, 115
operator (boolean), 150
Orienting Joints, 6, 8
Paint Weights, 123
paneLayout, 132
parser, 138
Particles, 63-65
Pick Walking Control Objects, 11
Pick Walking, 11
Placing Forearm Joints, 7
Placing Forearm, 6
Placing Joints, 6
plug-in (C++), 242
plusMinusAverage, 15
point cloud, 116
Pose Library, 129
pose space deformation, 213
Procedures, 25-26, 148
progress window, 169-170, 246
radians to degrees, 224, 225
recursion, 200, 204, 207
Rig Resoltion, 13
scalar, 221
Scale, 95
scope, (local vs global) 165
Sculpt Polygons, 119
Separation of Axes, 23
separator, 234
Set Driven Key (scripting), 129
setAttr, 31
setParent, 152
Shape node, 81, 237
Shape Set, 96, 115
single-axis driver, 221
skinCluster, 243
Smooth Binding, 120
Smoothing, 94-95
Soft body, 63-65
Soft Modification, 115
sourcing (MEL scripts), 148
Spine, 45
Springs, 63-65
square matrix, 226
Stiffness, 59
String, 26, 149
Subdivision Surface, 95
Substring, 76
tab layout, 234
text scroll list, 235
text, 152
timer (scripting), 205
tokenize, 175, 200, 245
Transform node, 4
Turbulence, 59
type cast, 170
user interface design, 130-134
Utility Nodes, 15, 40-42
variables, 149
Vector, 50, 197, 221, 222
Vertex Ordering, 116
white space, 152
windows command shell, 205
Wrap Deformer, 94
xform, 181
xpm image format, 131, 135

NOTES